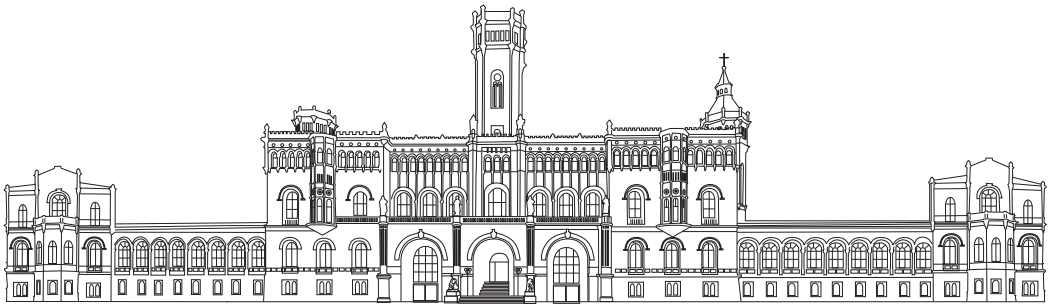


LEIBNIZ UNIVERSITÄT HANNOVER

Fakultät für Elektrotechnik und Informatik

On Construction, Performance, and Diversification for Structured Queries on the Semantic Desktop

Von der Fakultät für Elektrotechnik und Informatik
der Gottfried Wilhelm Leibniz Universität Hannover
zur Erlangung des akademischen Grades
Doktor-Ingenieur
(abgekürzt: Dr.-Ing.)
genehmigte Dissertation

von Dipl.-Inf. Enrico Minack

geboren am 15. Mai 1980 in Eisenhüttenstadt

2011

1. Referent: Prof. Dr. techn. Wolfgang Nejdl
2. Referent: Prof. Dr. Heribert Vollmer
Tag der Promotion: 14. Dezember 2011

Abstract The World Wide Web (the Web) has grown into a wide-spread everyday application. Since its invention in 1989, it experienced an inconceivable growth with fundamental impact on society and economy. With approximately two billion connected users and a trillion unique Web resources, the Web comprises the largest repository of general and expert knowledge. It is indisputably the largest publicly available source of human knowledge; notwithstanding the wide spectrum of quality, a remarkable achievement.

While the amount of information available on the Web reached an extent that renders it impossible for humans to process it in its entirety, the Web is still designed to be read and used by humans only. In contrast, computer systems with their enormous memory and processing capacities are still limited to only find Web content and provide humans access to it. The actual meaning of that content is still hidden from such systems.

As an extension to the Web, the Semantic Web provides technologies that allow enriching textual information and multimedia content with its actual meaning in an explicit encoding that is processable by machines. Then, computer systems can more effectively process Web content, combine pieces of information that are distributed among different Web sites into new knowledge, and provide intelligent services to humans or even solve tasks on behalf of them.

The Semantic Desktop utilizes such Semantic Web technologies on the Personal Computer Desktop. This allows users to explicitly articulate and access knowledge and the structure of information that they use in there everyday work and life. In this context, the dominant way of relocating and accessing data on the Desktop is search. In conjunction with Semantic Web technologies, classical Desktop Search reaches its limitations and consequently has to be extended to Semantic Desktop Search. However, this transition still poses challenges.

In this thesis, I contribute a Semantic Desktop search infrastructure that allows for easy integration of new semantic services. Further, this work identifies the inadequate performance of state-of-the-art components for this particular task. Based on these experiences, a number of improvements that target at different layers of semantic search are developed and evaluated throughout this thesis. In particular, this work provides contributions to the fields of Semantic Desktop search infrastructure, to the articulation, the evaluation, and the performance benchmarking of semantic queries, as well as result diversification that is applicable for Semantic Search.

Keywords Semantic Desktop, Semantic Search, Diversification

Zusammenfassung Das World Wide Web (das Web oder das Netz) hat sich zu einer weit verbreiteten alltäglichen Anwendung entwickelt. Seit seiner Erfindung im Jahre 1989 hat es ein unvorstellbares Wachstum erlebt und einen tief greifenden Einfluss auf die Gesellschaft und Wirtschaft entwickelt. Mit annähernd zwei Milliarden angeschlossenen Nutzern und einer Billion einzelner Web-Ressourcen stellt das Web die größte Ansammlung an Allgemein- und Spezialwissen dar. Es ist ohne Zweifel die größte öffentlich verfügbare Quelle an Wissen der Menschheit — trotz des großen Qualitätsspektrums eine bemerkenswerte Errungenschaft.

Obwohl die Menge an verfügbaren Informationen im Netz eine Größe erreicht hat, die es unmöglich macht, dass ein Mensch sie in seiner Gesamtheit liest, ist das Netz noch immer nur für das Lesen durch Menschen konstruiert. Ungeachtet der enormen Speicher- und Verarbeitungsmöglichkeiten von Computern wird ihr Einsatz immer noch darauf beschränkt, Menschen das Auffinden und den Zugang zu Informationen zu ermöglichen. Damit ist die eigentliche Bedeutung von Web-Inhalten solchen Systemen weiterhin versperrt.

Als eine Erweiterung des Webs bietet das Semantic Web (Semantische Web) Technologien zur Ergänzung von textuellen Informationen und multimedialen Inhalten um ihre eigentliche Bedeutung in der Art, dass diese Bedeutung von Maschinen verarbeitet werden kann. Dies ermöglicht es Computersystemen, Web Inhalte effektiver verarbeiten zu können, Informationsstücke, die über das Netz verteilt sind, zu neuem Wissen zusammenzusetzen, intelligente Dienste anzubieten oder sogar Aufgaben im Auftrag von Menschen zu erledigen.

Der Semantic Desktop (Semantische Desktop) verwendet ebensolche Semantic Web Technologien auf dem persönlichen Arbeitsplatzrechner (Desktop). Dies ermöglicht es den Nutzern, Wissen und strukturierte Informationen, die sie alltäglich im Beruf und Privatleben verwenden, explizit zu formulieren und wieder zu verwenden. Die dominierende Methode, um Daten wieder aufzufinden und darauf zuzugreifen, ist hierbei die Desktopsuche. In Verbindung mit Semantic Web Technologien stößt die klassische Desktopsuche allerdings an ihre Grenzen und muss konsequenterweise zur Semantischen Desktopsuche erweitert werden, was jedoch noch immer mit Herausforderungen verbunden ist.

In dieser Dissertation entwickle ich die Infrastruktur einer Semantischen Desktop Suchmaschine, die eine einfache Integration neuer semantischer Dienste ermöglicht. Darüber hinaus identifiziert diese Arbeit die unzureichende Leistungsfähigkeit modernster hierzu notwendiger Komponenten. Basierend auf diesen Erfahrungen entwickle und evaluiere ich Verbesserungen die auf verschiedene Ebenen der Semantischen Suche abzielen. In Speziellen leistet diese Arbeit einen Beitrag in den Bereichen Semantischer Desktopsuch-Infrastrukur, der Formulierung, Ausführung und Performanz Semantischer Suchanfragen, sowie der Diversifizierung von Suchergebnissen, wie sie auch in der Semantischen Suche anwendbar ist.

Schlüsselwörter Semantischer Desktop, Semantische Suche, Diversifizierung

Acknowledgements This work would not have been possible without the support and contribution of numerous people. I would like to take the opportunity to thank those that made me capable of creating this Ph.D. thesis.

First and foremost, I want to thank my supervisor Prof. Dr. techn. Wolfgang Nejdl for giving me the opportunity to work at the L3S Research Center, for his support, and invaluable advice. My thanks are also extended to Prof. Dr.-Ing. Gabriele von Voigt and Prof. Dr. Heribert Vollmer for dedicating their time and supporting my graduation as members of the examination board.

My time working at the L3S Research Center has been inspired by extraordinary people that perform great research. I am especially grateful to my mentor Dr. Wolf Siberski for his guidance over the past years and his contributions to this work. I am also very thankful for the fruitful collaboration with my co-authors, in particular (in alphabetical order) Ekaterini Ioannou, Gianluca Demartini, Gideon Zenz, Gunnar Å. Grimnes, Julien Gaugaz, Leo Sauermann, Paul-Alexandru Chirita, Raluca Paiu, Stefan Siersdorfer, Stefania Costache, and Xuan Zhou.

The person who had to bear the heaviest burden is my wife Kim Bartke-Minack. I owe my deepest gratitude to her for the enduring motivation, countless discussions, and for bringing me so often back on track to complete this thesis. I am also deeply grateful to my parents who provided me with the curiosity, determination, and drive that enabled me to take this journey.

I also want to thank Dr. Christian Bruns and Dr. Sven Lämmermann who inspired me to undertake the journey of graduation and particularly supported me to come to L3S.

I warmly credit all open source tools that I used throughout this thesis. Without the innumerous altruistic contributors of these projects, this Ph.D. thesis would have been much more labor-intensive if not even impossible to this extent. In particular, my thanks are dedicated to the following open source and free software projects: LaTeX, BibTeX, Kile, Tomboy, TaskCoach, Gnuplot, Dia, Perl, Eclipse, Subversion, Gnome Beagle, VirtualBox as well as Linux in general and Ubuntu in particular. This thesis uses icons created by Jojo Mendoza and Aha-Soft, published under the Creative Commons License at IconArchive.com.

This work was supported by the *NEPOMUK* project funded by the European Commission under the 6th Framework Programme (IST Contract No. 027705) and the FP7 integration project *LivingKnowledge* (Contract No. 231126). Some experiments were conducted on the high performance computing cluster of the *Regional Computing Centre for Lower Saxony* (RRZN).

"Some of our ideas have a natural correspondence
and connection with one another . . . "

— John Locke in *Of the Association of Ideas*, 1700.

"(The human brain) operates by association. With one item in its grasp, it snaps instantly to the next that is suggested by the association of thoughts, in accordance with some intricate web of trails carried by the cells of the brain."

— Vannevar Bush in *As We May Think*, The Atlantic Monthly, 1945.

"If HTML and the Web made all the online documents look like one huge book, RDF, schema, and inference languages will make all the data in the world look like one huge database."

— Tim Berners-Lee in *Weaving the Web*, 1999.

"And by design, anything we can talk about can be assigned a URI"

— T. Segaran, C. Evans, J. Taylor in *Programming the Semantic Web*, 2009.

"Where can I find more about that idea? Does it have a Wiki page?
An idea has to have a URI!"

— Leo Sauermann via Skype, 2007.

"Nothing shocks me. I'm a scientist"

— Harrison Ford in *Indiana Jones*.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Over the last decade, the growing amount of information available on the *World Wide Web* (called the Web) [24, 25] fueled the success of Web search engines [12, Ch. 13]. For the first time, users were enabled to locate Web pages via *search* that they would not have been able to find by *navigation*. Such pages became accessible by simple articulation of information needs: keyword queries. Due to the simplicity, this textual representation is inherently ambiguous. Consequently, these search engines still ignore the actual meaning of queries and Web pages.

A large proportion of the information provided by the Web is stored in databases in a structured way. When it is rendered into a visual presentation (this part of the Web is called the Deep Web [22]), this structure that complements the information with explicit meaning gets lost or at best becomes implicit in the rendering process. The remaining part of the Web is composed of textual or multimedia content that is inherently unstructured. Both, the unstructured information and the implicit structure are not easily processable by a machine. Apparently, the Web was built for human readers only.

In fact, the Web was envisioned to be machine processable [23, 26]. This allows computer systems not only to support humans in *accessing* pieces of information, but also in *processing* them. For this, such systems require means to access and exploit the meaning of information: the semantics. Such a Web where information contained in a Web page can be interpreted by a machine without any human ado is called the *Semantic Web*. With this, information from two different pages can be combined to deduce new knowledge. The goal of the Semantic Web is to overcome the isolated information silos that are created by those Deep Web databases [29]. If their structured information can easily be accessed and combined with other sources, the whole Web will become a machine-processable large knowledge base [23, p. 186].

Unfortunately, the lack of existing semantic information on the Web prevents the emergence of semantic applications. This in return causes a vicious circle since semantic applications are needed to motivate the publication of semantic information. Economically spoken, there is no benefit for one website to provide all its knowledge to be accessible, while all other websites that do not publish knowledge could already monetize this circumstance. Consequently, applications will hardly emerge if there is no critical mass of accessible information available. Further, users may have difficulties in understanding the benefits of Semantic Web technologies and how to use them if there are no semantic applications.

We can learn a possible solution to this dilemma from the *Social Web* (Web 2.0) [71, Sec. 1.3.2]. In the beginning of the Web, information only propagated from few producers to many consumers. Today with Web 2.0, however, Web users generate content themselves and consume content generated by others. When this user generated content employs Semantic Web technologies to explicitly encode their content's meaning, we will experience the bootstrap of the

Semantic Web. Since that content is usually generated on Personal Computers (PCs, Desktop Computers or Desktops), these should be semantified first so that the Semantic Web can emerge from that perimeter of semantic devices.

This idea motivates the *Semantic Desktop* [162, 163], where users can explicitly express and easier access their knowledge on their PCs. They can further publish this knowledge on the Semantic Web, interconnect their Semantic Desktops [62], and in return integrate knowledge from the Semantic Web with their Semantic Desktop. The successful creation of this Web of Semantic Desktops will foster the emergence of the Semantic Web.

Today, people use computers for their work and leisure activities, or to communicate with their social environment [31]. Desktop computers can store a lifetime of information [77], so users have to manage large volumes of heterogeneous data. Documents, images, or e-mails needed for a particular task are usually accessed via navigation or search. Where the former requires significant cognitive load for orienteering and remembering [181], search quickly provides the user with relevant results. Semantic information about the Desktop content can further improve Desktop search [122], while its realization however still poses challenges that prevent wide adaptation of this technology.

This thesis addresses some of these challenges, in particular those concerning efficiency and usability of such systems, with a focus on practical solutions and prototypical implementations. To deliver a convenient and efficient Semantic Desktop search experience, improvements at each layer of Semantic Desktop Search are investigated and presented in this work. This comprises:

1. A flexible Semantic Desktop search engine infrastructure that uses Semantic Web technologies and allows for easy integration of new semantic services for semantic information extraction, enrichment, and search. Further, a user study investigates the impact of the semantification of search on user satisfaction and search quality.

2. A simple yet powerful way to articulate structured keyword queries that allows users to fully exploit the semantics of their Desktop data without the need to know any query language or data ontology.

3. An approach that combines full-text search with structured search, as well as four particular in-depth performance improvements for the structural part of such a query evaluation.

4. The first full-text search benchmark for the Semantic Web. It comprises a synthetic data generator and a workload of 21 hybrid queries that allow to investigate the feature richness and evaluation performance of semantic stores.

5. A search result diversification approach that efficiently works for very large sets and naturally fits the task of diversifying streams of results as they exist in continuous query systems such as news, blogs or tweet retrieval.

These advances in Semantic Desktop Search allow for stronger adoption of the Semantic Desktop by users. With the help of the Semantic Desktop, we will see the Semantic Web emerge, and from Semantic Desktop search we will learn how Semantic Web search engines will provide access to this new Semantic World Wide Web — the Web 3.0.

Thesis Structure

In this Ph.D. thesis, I[1] study how Semantic Web technologies can be employed on the Desktop for the user task of searching personal data and knowledge. For this purposes, I develop a Semantic Desktop search infrastructure that bases on an open source Desktop search engine.

Figure 1.1: Structure of this Ph.D. thesis.

The development and usage of this Semantic Desktop search engine reveals the need for improvements of particular components of *Semantic Search*. This comprises the *construction* of semantic queries and their efficient *evaluation*. The performance of semantic search can only be measured and compared among different systems with the help of an appropriate *benchmark*. The user satisfaction regarding final search results can be improved with *search result diversification*, for which I propose a new scalable and efficient method. These fields of research relate to each other as depicted in above Figure 1.1.

This work comprises contributions of the following of my publications[2], organized into those five fields of research depicted in the figure above. Publications are given for each field in reverse chronological order and each of the subsequent chapters is dedicated to one of these fields.

[1]The usage of plural personal pronouns in scientific writing is common style. However, throughout this thesis I will use singular to stress my individual contribution. I may further stick to singular also in case of collaboration, where I then refer to significant contribution made to the collaborative work.

[2]One publication is listed twice as it contributes to two fields of research.

Semantic Desktop Search (Chapter 3 on page 29) The Semantic Desktop comprises more than semantic applications that run on a classical Desktop system. It requires a holistic integration of user data across application borders, though data are still to be created and managed with different applications [105, p. 52]. A Desktop-wide semantic infrastructure should be easily accessible for semantic applications, integrate data via common and extensible ontologies, and allow for free choice of programming languages.

A Semantic Desktop search engine, as an instance of semantic applications, manages a large volume of semantic data in order to provide access to these data via search. Such a system is an ideal platform for semantic services that process semantic user information in order to provide intelligent operations and create more knowledge. A Semantic Desktop search engine should be extensible by new semantic services to allow for quick integration and use. Existing systems are in fact difficult to extend or reuse.

Used as the basis for this thesis, with my contribution to the development of the semantic infrastructure project NEPOMUK[3] [131], developers are enabled to avoid application-specific information silos. Data are extracted from vendor-specific file formats and integrated in a central semantic store with well documented extensible ontologies and programming language independent access via standardized inter-process communication mechanisms [123].

We further employ this infrastructure for our extensible Semantic Desktop Search application that is successfully used as a platform for a number of research activities at the L3S research center [47–49,64,177,178]. With this prototype, we demonstrate that exploiting explicit structure of users' data for search improves search result quality and user satisfaction. This Semantic Desktop search engine is freely available as open source[4].

1. E. Minack, R. Paiu, S. Costache, G. Demartini, J. Gaugaz, E. Ioannou, P.-A. Chirita, and W. Nejdl. Leveraging Personal Metadata for Desktop Search: The Beagle^{++} System. *Journal of Web Semantics*, 8(1):37–54, 2010.

2. S. Chernov, E. Minack, and P. Serdyukov. Converting Desktop into a Personal Activity Dataset. In *Proceedings of 8th Nationational Russian Conference on Digital Libraries (RCDL'07)*, Pereslavl, Russia, Oct. 15–18, 2007.

3. S. Handschuh, T. Groza, K. Möller, G. Grimnes, L. Sauermann, M. Jazayeri, C. Mesnage, E. Minack, G. Reif, and R. Guðjónsdóttir. The NEPOMUK Project — On the way to the Social Semantic Desktop. In *Proceedings of iSemantics 2007*, Graz, Austria, Sep. 2007.

4. I. Brunkhorst, P.-A. Chirita, S. Costache, J. Gaugaz, E. Ioannou, T. Iofciu, E. Minack, W. Nejdl, and R. Paiu. The Beagle^{++} Toolbox: Towards an Extendable Desktop Search Architecture. In *Proceedings of the Semantic Desktop and Social Semantic Collaboration Workshop (SemDesk'06) at the 5th International Semantic Web Conference (ISWC)*, Athens, GA, USA, Nov. 6 2006.

[3] NEPOMUK — The Social Semantic Desktop: `http://nepomuk.semanticdesktop.org`.
[4] L3S *Beagle*$^{++}$ — Semantic Desktop Search: `http://beagle.l3s.de/`.

Semantic Query Construction (Chapter 4 on page 59) A Semantic Desktop Search application allows to exploit rich semantics for search. The user can implicitly or explicitly express structural properties of her information need in four different ways: *Keyword queries* with structural annotation only allow for simple structure, though can quickly become cryptic to read. *Natural language* is expressive and can encode the full range of knowledge structure, but such queries can be ambiguous and challenges a system to processes them. With *controlled languages*, expressive, generally valid and easily processable queries can be articulated. *Structural query languages*, finally, are primarily designed for application developers for comprehensive data access purposes and therefore, comprise complex syntax, are error-prone, and typify the complement to natural languages.

Due to simplicity in usage and implementation, among 32 Semantic Search applications surveyed in [94] the majority provides keyword search facilities. In contrast, there is no system that allows for easy articulation of explicit and expressive structure. Systems that provide this type of query articulation [8,32,61,90] require the user to be knowledgeable about structured query languages or data ontology. Obviously, there is a lack of explicit structured query articulation that requires only minimum knowledge and is as intuitive as keyword queries. I consider graphical query construction interfaces with controlled language the best compromise between expressiveness and simplicity of usage. Therefore, I investigate the design of graphical structured query articulation where the user is guided to express ontology-compliant relations among keywords.

The main contribution of this chapter is a formal incremental structured query construction model that is then implemented as the core of a graphical user interface. The architecture allows to plug in different recommender algorithms that provide the user with useful query construction steps at any time. This interface is freely available[5] as part of the open source Social Semantic Desktop NEPOMUK [131] [166, Sec. 2.10.2].

1. G. Zenz, X. Zhou, E. Minack, W. Siberski, and W. Nejdl. From keywords to semantic queries — Incremental query construction on the semantic web. *Journal of Web Semantics* 7(3):166–176, 2009.

2. R. Kawase, E. Minack, W. Nejdl, S. Araújo, and D. Schwabe. Incremental End-user Query Construction for the Semantic Desktop. In *Proceedings of the 5th International Conference on Web Information Systems and Technologies (WEBIST'09)*, pages 270–275, Lisbon, Portugal, March 23–26 2009, INSTICC Press.

3. E. Minack, W. Siberski, G. Zenz, and X. Zhou. SUITS4RDF: Incremental Query Construction for the Semantic Web. In *Proceedings of the 7th International Semantic Web Conference — Posters & Demos (ISWC'08)*, Karlsruhe, Germany, Oct. 26–30.

[5] NEPOMUK Eclipse: `http://nepomuk-eclipse.semanticdesktop.org/`.

Hybrid Query Evaluation (Chapter 5 on page 69) Once a user can articulate hybrid queries, they are to be evaluated against the semantic data store (RDF Store). Existing full-text search capable RDF stores do either not make use of the full potential of full-text search [37, 88, 89, 91, 149, 188], use non-standard query languages [89, 91, 149, 188], or have inconsistent semantics [37, 188]. Furthermore, existing implementations lack efficiency when supporting such queries, in particular for complex queries [125]. Therefore, high-performance full-featured full-text search for RDF that integrates with standard query languages is still needed.

As the next contribution of my thesis, I combine a widely used and well developed semantic store with an equally sophisticated keyword search engine, which allows for high-performance full-featured structured keyword query evaluation. A major part of this chapter presents in-depth store-specific enhancements made at different layers of the semantic store architecture. Finally, various performance evaluations verify the efficiency of this solution.

1. G. Zenz, X. Zhou, E. Minack, W. Siberski, and W. Nejdl. From keywords to semantic queries — Incremental query construction on the semantic web. *Journal of Web Semantics* 7(3):166–176, 2009.

2. E. Minack, L. Sauermann, G. Grimnes, C. Fluit, and J. Broekstra. The Sesame LuceneSail: RDF Queries with Full-text Search. *NEPOMUK Technical Report 2008-1*, Feb. 2008.

Benchmarking Hybrid Queries (Chapter 6 on page 103) Developers of semantic applications have the growing need for semantic stores that provide full-text search capabilities [94]. With a number of stores that provide this functionality, developers are obligated to pick the store that best fits their needs. However, to be able to compare the performance of full-text capabilities of RDF stores, these need to be measured in a repeatable and comparable way, which can only be done with a publicly available benchmark.

Since at the time of this work, no benchmark for semantic keyword query evaluation exist, I further conduct a performance study of RDF Store keyword search capabilities. For this, I develop a benchmark that comprises a synthetic scalable RDF data generator with text data that exhibit realistic word distributions. Additionally, queries that target at particular hybrid query evaluation challenges are created. On the whole, four RDF stores are evaluated, revealing individual characteristics.

1. E. Minack, W. Siberski, and W. Nejdl. Benchmarking Fulltext Search Performance of RDF Stores. In *Proceedings of the 6th European Semantic Web Conference (ESWC'09)*, pages 81–95, Heraklion, Crete, Greece, May 31–June 4, 2009.

Diversifying Search Results (Chapter 7 on page 117) In case a query produces a long list of relevant results, users are confronted with a situation called information overload [7, 99]. In order to increase user satisfaction and decrease user's cognitive load, result diversification is employed. This technique aims at providing the user with a short list of relevant but also diverse results that best represent the entire list of results.

For structured and unstructured data in general, diversification of search results for large result sets or streams of results is expensive by means of computation or consumed memory. A memory efficient diversification of large sets and a computationally efficient algorithm for diversification of result streams is still missing.

As a last contribution of my work, I present our new approach of efficient and effective result diversification. The input set of results is processed in a streaming-fashion, which allows for diversification of large sets and streams. An extensive evaluation on three real-world data sets shows that this approach does not suffer from diversification quality while exposing significant improvements on memory consumption and computational complexity.

1. E. Minack, W. Siberski, and W. Nejdl. Incremental Diversification for Very Large Sets: a Streaming-based Approach. In *Proceedings of the 34th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR'11)*, July 24–28 2011, Beijing, China.

2. D. Skoutas, E. Minack, and W. Nejdl. Dealing with Diversity in Web Search Results. In *Proceedings of the 2nd ACM International Conference on Web Science (WebSci'10)*, Raleigh, NC, USA, April 26–27 2010.

3. E. Minack, G. Demartini, and W. Nejdl. Current Approaches to Search Result Diversification. In *Proceedings of The 1st International Workshop on Living Web at the 8th International Semantic Web Conference (ISWC'09)*, Chantilly, VA, USA, Oct. 25–29 2009.

Furthermore, the following publications have been published throughout my work as a Ph.D. student at the L3S Research Center, but do not further contribute to this thesis:

1. S. Siersdorfer, J. Hare, E. Minack, and F. Deng. Analyzing and Predicting Sentiment of Images on the Social Web. In *Proceedings of the 18th ACM International Conference on Multimedia (Multimedia'10)*, Oct. 25–29th, 2010, Firenze, Italy.

2. P. Zontone, G. Boato, J. Hare, P. Lewis, S. Siersdorfer, and E. Minack. Image and Collateral Text in Support of Auto-annotation and Sentiment Analysis. In *Proceedings of the 5th TextGraphs Workshop at the 48th Annual Meeting of the Association for Computational Linguistics*, July 16 2010, Uppsala, Sweden.

Chapter 2

The Semantic Web on User Desktops

It has been a long scientific endeavor to create the *World Wide Web* (the Web) and the Semantic Web. An integral property of the Web can already be found in a seminal article from 1883 [58]. Charles A. Cutter describes a future library where people can request a particular book electronically via a telephone network while the physical delivery is done fully automatically to the person's desk. Such an electronic request of documents is very similar to the way Web pages are accessed today. Half a century later, Vannevar Bush introduces the notion of linking pieces of information on his MEMEX desk, an early Hypertext System [33, 55] and Semantic Desktop [35]. These links represent associations, the fundamental principle of how the human mind manages knowledge [187, 190]. Such associations among things can be considered as what is 50 years later called relations among entities or resources [26] [171, p. 65] in the context of the Semantic Web. With the advent of computer networks in the 70s and 80s of the past century, inter-linked electronic documents could be requested and delivered from distributed locations [23, 137, 138]. These hypertext systems allowed for editing and publishing electronic documents, connecting pieces of information with typed or untyped, uni- or bi-directed links.

Though a number of different hypertext systems had been developed by that time [33, 55], only the Web has grown into a wide-spread everyday application. Since its invention in 1989, it experienced 20 years of inconceivable growth with fundamental impact on society [39] and economy [96]. This success originates from three fundamental design principles [23, Ch. 4] [24]:

Universality The Web works for any kind of computer system, display and processing capabilities, over any kind of computer network, for any kind of data and for everybody.

Neutrality The neutrality principle refers to the communication technology underlying the Web, the *Internet*. It should not behave differently *w. r. t.* the requested services, the information accessed, or the individual users.

Decentrality Based on three open WWW standards, everyone can participate and contribute to the Web without asking a central authority for permission. These standards are royalty-free, transparently reviewed and accepted by experts, and constantly extended to evolving needs of the users of these standards.

The three open WWW standard protocols are the Hypertext Markup Language[1] (HTML) used for Web content layout annotation and linkage, the Uniform Resource Identifier[2] (URI) used for unique identification of Web content, and the Hypertext Transfer Protocol[3] (HTTP) that is used for Web content retrieval.

[1] RFC 1866 – HTML: `http://www.ietf.org/rfc/rfc1866.txt`.
[2] RFC 2396 – URI: `http://www.ietf.org/rfc/rfc2396.txt`.
[3] RFC 2616 – HTTP/1.1: `http://www.ietf.org/rfc/rfc2616.txt`.

2.1 The Semantic Web

The invention of the Web by Tim Berners-Lee is considered as much a media revolution as the invention of modern book print by Johannes Gutenberg in the 15th century[4]. The amount of information available on the Web is hard to measure and, due to the dynamic nature of the Web, inherently outdated. Estimates vary from several billions of Web pages [85] to one trillion URIs known to exist on the Web[5]. As of June 30, 2010, almost 2 billion people use the Internet and Web today[6]. Indisputable, the Web comprises the largest repository of general purpose, as well as specific expert information. It is the largest publicly available source of human knowledge.

This huge and ever growing amount of information is primarily consumed by humans. Computer systems are only used to provide access to information, and to support users in finding, processing, and managing large volumes of information. What computers cannot sufficiently do today is to consume Web content autonomously in order to perform intelligent tasks for humans or to provide intelligent services. Computers cannot, for instance, combine information from different sources in order to create new knowledge. Certainly, automatic tools exist that can extract knowledge and could feed such a system [129, 130, 145], however, such solutions are noisy, domain-dependent, require domain experts, and are computationally intensive. In that sense, humans still have to read and understand Web content themselves.

This challenge was early identified and addressed by Tim Berners-Lee and his idea of the Semantic Web [26]. With Semantic Web technologies, information published on the Web can be semantically annotated in a machine-processable way. There are five fundamental Semantic Web technologies envisioned for this extension. We recommend [6, 59, 171] as a complete reference.

Metadata Semantics of information can be explicitly articulated and encoded as *metadata*, *i. e.*, data about data. This allows to enrich textual representations of information and knowledge by explicit structures. For instance, the textual information `"John Smith"` can be extended to

```
<Person>
  <firstname>John</firstname> <lastname>Smith</lastname>
</Person>
```

Here[7], the textual labels in angle brackets (*e. g.*, `<firstname>`) denote semantic annotations called *tags*, whereas a pair of tags (`<...>` and `</...>`) encloses other tags or textual values (*e. g.*, `John`). The example metadata state that there is a `Person` with `firstname "John"` and `lastname "Smith"`. In contrast to

```
<Company><name>John Smith and Sons</name></Company>
```

a computer system immediately knows the difference between both pieces of information more easily than without semantic annotations. A framework that allows to express such semantics is the Resource Description Framework (RDF) [83]. The cornerstones of RDF are *resources* and

[4]The TIME Magazine: Tim Berners-Lee: The Man Who Invented The Web: `http://www.time.com/time/magazine/article/0,9171,986354,00.html`.

[5]Official Google Blog: We knew the web was big...: `http://googleblog.blogspot.com/2008/07/we-knew-web-was-big.html`.

[6]Internet World Stats: `http://www.internetworldstats.com/stats.htm`.

[7]This is an example notation for semantic information based on XML notation.

relations among resources, both uniquely identified by URIs, and values called *literals*. Such literals can denote to any type of value that has a textual representation (*e. g.*, texts, numbers, dates). RDF metadata are composed of *statements*. An RDF statement consists of *subject*, *predicate*, *object*, and an optional *context*. The subject is always a resources, a predicate is a relation, the object can be a resource or a literal, and the context is a resource. The above example in RDF statements (without a context) looks like

```
ex:johnsmith rdf:type ex:Person .
ex:johnsmith ex:firstname "John" .
ex:johnsmith ex:lastname "Smith" .
```

where `ex:johnsmith` refers to a particular resource and `rdf:type` denotes that this is a (is an instance of *class*) `ex:Person`. URIs in this example are given in abbreviated notation. The prefixes `ex:` and `rdf:` refer to namespaces `http://example.com/` and `http://www.w3.org/1999/02/22-rdf-syntax-ns#`. Therefore, the predicate of the first statement resolves to `<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>`, while subject `ex:johnsmith` resolves to `<http://example.com/johnsmith>`. An in-depth introduction of RDF framework can be found in [6, Ch. 3], [59, Ch. 5], and [171, Ch. 4].

Ontology An ontology defines a vocabulary of such tags, for instance, `<firstname>` or `<http://example.com/Person>`. Ontologies can be defined using ontology languages such as RDF Schema (RDFS) [82], DAML+OIL Web Ontology Language (DAML+OIL) [56], or the Web Ontology Language (OWL) [81]. All these ontology languages allow for class (*e. g.*, a person), relation (*e. g.*, the first name of a person), sub-class, and sub-relation definitions. A relation is defined by a URI, a domain, and a range. The latter two restrict resources to be of particular types in order to be used as subject (domain) or object (range) of a statement. The `ex:firstname`, for instance, can only be used for a person as domain and a literal as the range. Sub-class and sub-relation definitions form a hierarchy of generalization and specialization of classes and relations.

DAML+OIL and OWL languages add richer modeling primitives to RDFS, which allow for more powerful ontologies that on the other hand can be computationally more expensive. For further details on these ontology languages, please see [6, Ch. 4], [59, Ch. 8], and [171, Ch. 6].

Logic With the help of formal semantics (*i. e.*, *logic*) defined by the employed ontology language[8,9,10], implicit knowledge of that ontology, as well as knowledge encoded with this ontology, can be made explicit by inference (also called reasoning). For example, a `<friend-of>` relation can be defined to relate two people to each other (ontology level). Given that A is a friend of B (metadata level) a computer system can deduce from domain and range of the relation that A and B are people (schema level). A common task for reasoning is to compute the complete sub-class and sub-relation hierarchy and all `rdf:type` information induced by these. This is called the *full transitive closure*. Please further see [6, Ch. 5], and [59, pp. 226–229].

[8] RDF Semantics: `http://www.w3.org/TR/rdf-mt/`.
[9] OWL2 — Direct Semantics: `http://www.w3.org/TR/owl2-direct-semantics/`.
[10] A Model-Theoretic Semantics for DAML+OIL: `http://www.w3.org/TR/daml+oil-model`.

Query Languages We have seen means to encode semantic information in the standardized data model RDF, to articulate it in well-defined RDFS ontologies, and to entail it based on logic and inference, but means of data retrieval are still missing. With structured query languages such as RDQL [170] or SPARQL [146], structural properties of the data to be retrieved can be specified in the same way as RDF models the data itself, that is as statement. The subject, predicate, object, or context of a statement can arbitrarily be left unspecified by a wildcard, and actual values of instantiations of such statement patters can be retrieved via variables.

Following our example from above, the given RDF data can be queried for all people's first and last names using this SPARQL query:

```
SELECT
  ?s, ?fn, ?ln
WHERE {
  ?s rdf:type ex:Person .
  ?s ex:firstname ?fn .
  ?s ex:lastname ?ln .
}
```

This retrieves the URIs, first and last names of all people, whereas, *e. g.*, no company names will be retrieved. The variable bindings of this query evaluated against the three statements above can be organized in a table as:

?s	?fn	?ln
ex:johnsmith	"John"	"Smith"

Query results can also be retrieved in form of an RDF graph, where the above variable bindings represent an intermediate result used to instantiate graph construction statement patterns. The result of such a `COSTRUCT` query

```
CONSTRUCT {
  ?s rdf:type ex2:Person .
  ?s ex2:givenname ?fn .
  ?s ex2:surname ?ln .
} WHERE {
  ?s rdf:type ex:Person .
  ?s ex:firstname ?fn .
  ?s ex:lastname ?ln .
}
```

is a sequence of RDF statements:

```
ex:johnsmith rdf:type ex2:Person .
ex:johnsmith ex2:givenname "John" .
ex:johnsmith ex2:surname "Smith" .
```

With such a query, the sub-graphs that match the statement patterns of the `WHERE` clause can easily be transformed into a different graph (*e. g.*, with a different ontology or structure) defined by the `CONSTRUCT` clause.

For more practical details on the SPARQL query language, we recommend [171, pp. 86–96].

Agents Having metadata, ontologies, logic, and query languages at hand, autonomous software systems called *agents* can intelligently perform tasks on behalf of humans [26, 93] [59, p. 245]. When information is published on the Web and encoded in ontologies known to the agents, such agents can access and process such information easily. An example of such tasks is answering questions such as:

`"Which German cities have more than one million inhabitants?".`

The agent can use Semantic Web search engines [66, 95] or Semantic Web service registries [59, p. 245] to find Semantic Web content and services that provide relevant fragments of information (cities in Germany, the number of inhabitants of cities), query these documents and integrate partial answers into the final list of results. Alternatively, all relevant documents can be downloaded and integrated in one local semantic data store and that is then queried with the user query.

Fortunately, a large portion of Web content is already structured and stored in databases. These data are rendered into Web pages as textual representation. Examples are product catalogs or user profiles. Currently, during this step, existing structure of the data gets lost or at least becomes implicit. With Semantic Web technologies, this structure can be preserved, put online in an explicit format, and made available to semantic agents.

Though necessary technologies are available, their general adoption by the public (private and corporate entities) is still missing. Companies and commercial website operators require economic benefits to invest into such technologies in order to get them publish their currently hidden knowledge. Unfortunately, as long as no one provides semantic information, semantic applications cannot emerge or develop their full potential.

The other portion of Web content can be characterized as pure textual or multimedia content. Most of this content resided, before being published on the Web, on a desktop computer [162, p. 6]. Such devices can be seen as the outer perimeter of the curremt Web. From there, a large fraction of content migrates into the Web.

This perimeter is the point where the Semantic Web and its emergence starts [163]. With the semantification of such devices, semantic information can start to propagate into the Web and semantic applications can emerge upon them. These devices are usual desktop PCs, whereas their semantified variants are called Semantic Desktops [163].

2.2 The Semantic Desktop

Apart from that potential to bootstrap the Semantic Web, the Semantic Desktop [163] is motivated by the observation that an increasing amount of information is handled and tasks are performed with computers. With the growing capacity of hard disks, data do not have to be erased anymore. A lifetime of data can be stored forever [77].

Consequently, over the last one or two decades, a significant part of information relevant for people's work[11] and life happened to be stored on Computer Desktops [31]. As in Vannevar Bush's 1945 vision of the MEMEX [35], the Computer today helps to store, locate, consume,

[11] Here we refer to so called *knowledge workers*. These are people that consume and aggregate information and knowledge to create new knowledge. Examples are, among many others, scientists, engineers, or managers.

and manage tremendous amounts of data, such as correspondence (e-mail, instant massaging, written letters), information (web pages, manuals, articles) as well as non-textual media such as images, videos or music [77].

The management of these data is called *Personal Information Management* (PIM). This comprises the acquisition, organization, maintenance, and retrieval of digital information (*e. g.*, e-mail, files, contacts) by an individual in a personal (work and home) computing environment [31]. The difference between acquisition and retrieval of information is that whereas the former refers to *finding* new information, the latter denotes the action of *re-finding* previously seen information that was stored within the PIM system. In this thesis, I focus on the retrieval task of Personal Information Management.

Data used with different applications are stored in application-specific information silos, separated from other applications and each potentially providing its own organizational scheme. The file system, for instance, stores all kinds of files in a hierarchical manner. The path to a file can be used to categorize the content of a file. While, the content of a file can be categorized along different orthogonal dimensions, the file itself can only be put under a single path. The same holds for e-mail clients or bookmarks. Though all employ the same organizational model (here a tree-like hierarchy), data items as files, e-mails and bookmarks are all isolated and cannot be put into a comprehensive context. In other words, the user cannot have e-mails along with relevant files or bookmarks in the same folder, though they would greatly benefit if this would be the case [105, p. 52].

Even more interestingly, the user of all these data items has a large amount of knowledge about them: who is depicted on a photo, where and when was it taken, how strong is the relationship with a person, where did the user meet somebody, *e. g.*, the sender of an e-mail. Thus, the knowledge of the user about the data stored on the Desktop extends his personal information and is currently not explicitly stored on the computer, but in the user's mind, and can therefore not be used by the computer system to support the user. Therefore, the knowledge has to manifest on the Desktop, the Semantic Desktop.

However, as data items are isolated by application borders, also the knowledge about them would then be separated. A holistic retrieval and integration of data from multiple isolated stores require in worst case a Many-to-Many integration and at least a Many-to-One solution [162, p. 7]. There are a couple of strategies to overcome this burden:

Monolithic Replacement A monolithic application that replaces all existing applications can integrate information of all these applications [147]. It certainly needs to provide access to the information, otherwise multiple small isolated silos are replaced with one large silo. However, users would be required to migrate to a new application which the majority of users might not be willing to do.

Semantic Infrastructure With the existence of a semantic infrastructure, application developers can store information in a central repository and provide access in a unified way. The application has full control to which extend information is published while no replication happens. Standardized ontologies provide a vocabulary shared among applications that allows for integration of these data so that all applications can make use of them. The NEPOMUK project implemented such an infrastructure [131].

Semantic Access Alternatively, applications could provide standardized interfaces that allow for identifying, representing, and retrieving data in these silos in a distributed fashion as done on the Semantic Web [29]. This strategy requires all applications to be modified, as well as establishing a distributed system of semantic stores on a single computer with means for discovery, access, and search. The primary advantage is that no replication of data happens while applications keep full control over their data.

Semantic Extraction The extension of existing applications can be laborious or even impossible for closed source software. This extension can be avoided when data containers of particular applications (e-mail client in-box files, the Web browser cache directory, files of various types) are accessed directly. Then, semantic information is extracted and integrated into a central semantic store. Since such data containers are encoded in application-specific ways, this method constitutes a Many-to-One approach. Further, data have to be replicated but on the other hand full access is gained over the data [122,164,189]. Since this approach allows accessing a large amount of data with no modification of the respective applications, I focus on this strategy in my thesis (see the next chapter).

Whichever strategy is chosen, once user knowledge is explicitly articulated and stored, the following advantages render possible:

Re-finding and navigating past knowledge: The human brain developed a strategy of filtering irrelevant or infrequently used information. In some cases, information gains relevance over time so it is forgotten when it is needed at a later point in time. In other cases, information may be irrelevant for the human to survive, but relevant for human to live. Due to evolution, the brain favors the former type of information and sometimes struggles to deem information of the latter to really be important enough to memorize.

Information has to be used or recognized repeatedly in order to convince the brain of its importance [187, p. 37]. Articulating information, knowledge, and experiences makes the human reflect on them, which improves information quality and provides the human brain with evidence of importance. In any case, with the help of the computer, the articulated and stored knowledge will never fade and can be re-found and navigated any time later on.

Sharing and reuse of knowledge: In a group of people like an online community or a corporate environment, the knowledge of one person is of high value for others. Everybody gains experience and acquires knowledge, every day. In order to avoid everyone repeating the same time-intensive learning processes, people make use of others' knowledge. For this, they have to find those people that have relevant knowledge. Making knowledge explicit for computer systems allows for reuse, search and shared access [18].

Furthermore, when people leave a company, they take all their implicit knowledge with them, moving it out of the scope of the company. Having such knowledge explicitly stored on computers, it can further be of use for the company.

Enrichment of Desktop information: The knowledge being articulated usually refers to information items stored on the Desktop, such as e-mails, person contacts, web pages, documents, *etc.*. These references between Desktop items create a network that can be used for navigation and ranking search results [50, 122, 134, 135].

In the NEPOMUK project[12] [131], we followed a hybrid strategy where we created a semantic infrastructure and bootstrapped the population of a central semantic store by the semantic extraction strategy. My work contributes to the extraction part of that hybrid strategy, complemented by means to search such data [76, 123], as reported in the next chapter.

[12]NEPOMUK — The Social Semantic Desktop: `http://nepomuk.semanticdesktop.org`.

Chapter 3

Semantic Desktop Search

We have learned from the previous chapter that once knowledge can be stored and articulated explicitly, new means of retrieving it can arise. As we will see, Semantic Desktop Search requires an enhanced infrastructure compared to classical Desktop Search. However, such an infrastructure still contains main building blocks of a classical Desktop Search system, and thus can benefit from this fact by reusing common parts that are established and well understood.

In this chapter, I elaborate on important components of such a Semantic Desktop Search infrastructure which I implemented for the *L3S Beagle*$^{++}$ prototype [122] — the semantification of the open source *Gnome Beagle* Desktop Search engine — and integrated into the NEPOMUK Social Semantic Desktop [76, 123, 131].

With this work, I make the following contributions:

1. I design an extensible and flexible architecture for a Semantic Desktop search engine in which components written in different programming languages and frameworks nicely integrate into our search engine. This engine bases on an existing classical Desktop Search engine and therefore reuses mature and established components and provides a blueprint how to semantify a classical Desktop search engine.

2. Our evaluation of the base system *Beagle* and its semantified version *Beagle*$^{++}$ allows to directly measure the impact of semantification on search result quality and user satisfaction. Results show that Semantic Search finds more relevant results. For example, precision among the top four results increases from 66 % to 89 % for clear queries. Structured queries even exhibit a precision of up to 92 % in our user study.

This work lays the foundation for numerous researchers at L3S to implement, integrate and evaluate their particular research work in a Semantic Desktop Search environment. The *Beagle*$^{++}$ search engine was further employed to feed other research projects with Semantic Desktop data [47–49, 64, 177, 178].

After a short introduction into the characteristics of Semantic Search in the next section, I review existing approaches in the area of Semantic Desktop Search in Section 3.2. Section 3.3 presents the flexible architecture of our search engine. The three building blocks metadata generation and storage (Section 3.4), metadata enrichment (Section 3.5) and metadata search (Section 3.6) constitute the Semantic Desktop search engine. The quality of search results is evaluated in Section 3.7 with a user study. I deliberate on lessons learned and consider the public perception of this work in Section 3.8. The work presented in this chapter can be seen as the seed and motivation for the work presented in the remainder of this thesis. As a conclusion of this work, I list a number of challenges identified during the implementation and usage of the *Beagle*$^{++}$ search engine in Section 3.9.

3.1 Introduction

Once the user has stored her data and knowledge on computer systems, she needs quick access to relevant information to be able to perform a particular task. People usually perform this *Personal Information Management* (PIM) task called retrieval of digital information [31] using one of the following main strategies [181]:

Navigation: Structures like directory hierarchies (file system, e-mail folders) or navigational links (web pages) are used to classify information. Navigation becomes problematic when multiple locations within this classification are valid, but the information can only be put at one. This is usually the case for hierarchical structures like the file system or e-mail folders. Then, the information has either to be duplicated, which introduces redundancy and therefore the possibility of inconsistency, or it is only put at one of these locations. In that case, the user has to investigate all possible locations in order to re-find the information. Though a hierarchical structure is not necessarily the best structure to organize information, it is certainly a good organization structure in most cases [105, p. 21–24].

Search: Many people prefer search over navigation, because the cognitive effort of search is much lower. And those users that first try navigation usually fall back to search, either because no starting point for navigation comes to their mind or navigation simply does not provide satisfying results [105, p. 85].

For search, users only need to describe the required piece of information with a very short query and retrieve a list of relevant resources. Keyword queries are usually employed as the simplest way of searching because they are very easy to articulate. However, keywords are ambiguous and less expressive than those queries that also consider structural properties of target resources [12, Sec. 2.9].

Semantic Desktop Search bases on two areas of research that evolved separately over decades: Information Retrieval (IR) deals with unstructured data and the Database community addressing structured or semi-structured data. These two disciplines tend to more and more merge into one holistic theory of searching information [5, 10, 16].

The demand of integrated means to search structured and unstructured data is articulated by the large number of semantic applications that provide access to RDF data (structured data) via keyword searching capabilities (unstructured search). This was confirmed in the Semantic Web Survey [94], where the majority of 35 studied semantic applications provide full-text search. Where pure full-text search ignores the semantics of Desktop items and the relations among them, Semantic Search exploits both kinds of information: textual and structural.

Semantic Search In contrast to navigation that uses structure, and keyword search that targets at textual content, Semantic Search exploits structure to improve keyword search quality. With structural properties, queries become more expressive and produce smaller result sets in which results are generally more relevant [12, Sec. 4.4]. It is a hybrid of both strategies: searching with keywords while using short-range navigational constraints.

3.2 Related Work

The Semantic Desktop has grown into a vivid field of research over the last decade so that a number of semantic applications and Semantic Desktop projects predate this Ph.D. thesis [105]:

The Semantic Desktop The Fenfire project [74] proposes a solution to interlink any kind of information on one's Desktop. That might be the birthday with the person's name and an article she wrote. The idea is to make the translation from the current file structure to a structure that allows people to organize their data in a more flexible way that is similar to reality and to their personal needs. Unfortunately, this project is not based on WWW standards.

Haystack [147] is in some sense similar to Fenfire. This project emphasizes the relationship between an individual person and her corpus. It learns from user interaction with information which documents are relevant, collects metadata about them, and allows for manual annotation. Further, it establishes connections between documents that are similar in their content and then exploits all these pieces of information for browsing, searching, and accessing information. Unlike our work that focuses on searching data, their work primarily organizes such data.

A third project that builds an information management environment for the Desktop is Gnowsis [161]. This project envisions the possibility to make any kind of resource addressable on the Desktop and to link any two such resources with a semantic connection. The primary burden here is that such pieces of information are stored and managed by different applications in application-specific manners (see information silos in Section 2.2). Relevant bits of information become connected and accessible when they are required for a specific user task. In 2003, this project started with a prototype developed during a diploma thesis that consistently applied Semantic Web technologies on the user Desktop. Gnowsis matured during the Ph.D. thesis [162] of the same author to a full-featured Microsoft® Windows®-based Semantic Desktop infrastructure that integrates numerous office applications. While Gnowsis focuses on turning the Desktop into a Semantic Desktop, we focus on the Semantic Desktop Search functionality for such a Semantic Desktop in particular. As Gnowsis, we base our semantic application onto the same generic open semantic desktop architecture [123].

The authors of "The Social Semantic Desktop" [62] introduce the social entailment of the Semantic Desktop. They envision a collaborative application that combines Semantic Desktops and their Semantic Web technologies [161] with social networking infrastructures [104] and Semantic P2P technologies [136] in order to interconnect individual Semantic Desktops via the social network of their users. The purpose of this joint effort is to share and interconnect data and metadata, as well as to reuse classification and categorization efforts at a global scope. The Gnowsis project and this seminal paper constituted the seed for the European Project "NEPOMUK: The Social Semantic Desktop". Major parts of our work was created within that project, and NEPOMUK contributions were reused in this work in return.

Compared to those existing approaches, we share the same goals of connecting Desktop resources. But we additionally utilize these links in order to build a better search engine on the Desktop that combines traditional IR methods with the Structured Search, also allowing the user to experience and explore her metadata network.

Semantic Desktop Search Desktop Search is not a new technology, but the interest in it has been increasing over the last decade. As a result of the growing volume of data users store and use on their Personal Computer, all major Web search engines intensified their search services on desktop PCs. They have released several free Desktop search engines, *e. g.*, Google Desktop Search[1], Microsoft® Bing™ Bar[2], and Yahoo! Desktop Search[3]. Some providers even integrated Desktop search into their operating system[4]. The open source community also provides Desktop Search projects such as Beagle[5] for Gnome and Strigi[6] for KDE.

All these Desktop Search engines support a very exhaustive list of file types and consider explicit metadata contained in files. However, they model metadata as textual data — speaking in the notion of RDF: as literals. No entities such as people or locations are extracted and manifested as independent resources. Therefore, files are not connected to other files that share the same metadata or reference the same resources. These classical Desktop Search systems miss the opportunity to improve user experience by exploiting the rich context Desktop items provide [50, 134, 135]. This context is of value for users to relocate information as studies have shown [181]. With our semantified Desktop search engine, we present how this can be achieved and we evaluate the improvement in user satisfaction [122].

Several academic search and retrieval systems make extensive use of semantic relationships that can be inferred on the Desktop. For instance, in *Stuff I've Seen* [70], contextual cues like last file access time or author information are used to enrich search results. Swoogle [66] offers information retrieval capabilities for semantic documents that reside on the Web. It provides a ranking for RDF documents that employs different weights for individual types of semantic links to reflect the probability that users explore them. It integrates all ontologies that exist on the Web into their ranking scheme.

Another interesting semantic search tool is the TAP project [84]. It uses metadata extracted from information silos such as database-backed Web pages to enrich search results. Their semantic search results are independent of the traditional keyword search results and aim to augment them. Another semantic search method is proposed in [151]. It first performs a classical text-based search on metadata, whose output is then extended using the RDF network between semantic concepts. Finally it is reordered with techniques adapted from IR. In contrast to these systems, our work integrates the classical IR search with Semantic Search in one holistic step.

Structured Queries In the context of semantic querying, several languages to interact with a semantic database have been proposed. The most common language to query RDF is the W3C recommendation SPARQL [146]. Its main characteristic is that it allows to obtain an RDF graph as a query result encoded in two ways: 1) a table of variable bindings (with rows of variable values) for each existing instantiation of the structural pattern that is used in the query and 2) an RDF graphs constructed by the query based on those bindings.

[1]Google Desktop: `http://desktop.google.com/`.
[2]Bing™ Bar: `http://toolbar.msn.com/`.
[3]Yahoo! Desktop Search: `http://desktop.yahoo.com/`.
[4]Apple – Spotlight: `http://www.apple.com/macosx/what-is-macosx/spotlight.html`.
[5]Gnome Beagle: `http://www.gnome.org/projects/beagle/`.
[6]Strigi: `http://strigi.sourceforge.net/`.

The disadvantage of this language is that a user has to learn a complex query language that is error-prone. Therefore, in our work, we did not directly base the query interface on such a language. With our system, the user either types keywords in order to search for content, types queries annotated with simple structure, or constructs complex structured queries with a graphical user interface (presented in the next chapter). The system then translates the user query into the equivalent valid SPARQL query.

Another drawback of this language is that it does not support keyword search but only full string matching or regular expression filters, which are inherently inefficient. Our Semantic Desktop Search bases on a standard compliant full-text search extension of the SPARQL language. Chapter 5 on page 69 is dedicated to this particular challenge of combining full-text search with Semantic Search.

3.3 Semantic Desktop Search Infrastructure

In this section, I present the architecture of the *Beagle*$^{++}$ prototype. It defines the frame for the subsequent sections that describe the components of our Semantic Desktop Search prototype.

We build our Semantic Desktop Search application on an existing classical Desktop Search engine mainly because of two reasons. Firstly, both search engines have a lot of tasks in common. To list only a few, both need to monitor data containers for changed data items, schedule the extraction of full-text content stored in application-specific data formats, provide storage and query facilities, user interfaces for search and configuration, *etc.*. We did not aim at re-inventing the wheel in areas that are sophistically covered by open source software projects and can therefore easily be reused. Secondly, this strategy allows to us evaluate the impact of *semantifying* Desktop Search on search result quality and user satisfaction. For this, a competitive system for comparison is needed. The best comparable system is the one that itself gets semantified.

In the following, I give an overview of the semantic infrastructure that is used to semantify the base system, as described throughout this chapter.

3.3.1 Open Infrastructure for Semantic Applications

Within the NEPOMUK project [131, 150], we developed an infrastructure for semantic applications that brings Semantic Web technologies to the Desktop. It is based on Semantic Web Services that run on the Desktop to provide applications with semantic services. The infrastructure is designed in three layers as depicted in Figure 3.1. *Semantic applications* that deploy the infrastructure constitute the upper layer. These applications gain access to Semantic Web Services that run in the inner *Semantic Middleware* layer by open communication standards such as HTTP[7], OSGi[8], and D-Bus[9]. These technologies make this architecture independent of the employed hardware and programming languages. The lower layer contains functionality that interconnects Semantic Desktops; the essence that makes the Semantic Desktop the Social Semantic Desktop.

[7]RFC 1866 – HTML: `http://www.ietf.org/rfc/rfc1866.txt`.
[8]OSGi Alliance: `http://www.osgi.org/`.
[9]D-Bus: `http://www.freedesktop.org/wiki/Software/dbus`.

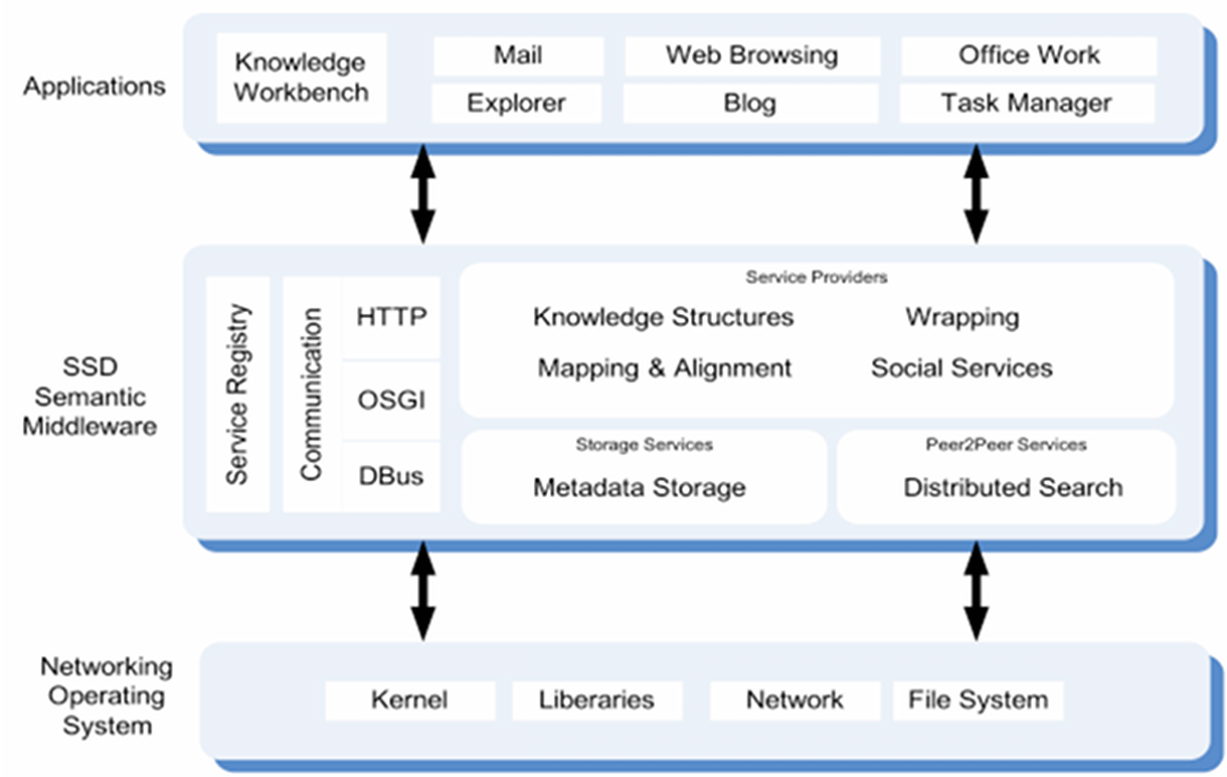


Figure 3.1: Three-layered architecture of the NEPOMUK semantic infrastructure.

Central services of the Semantic Middleware are giving structure to Desktop knowledge encoded in ontologies, ontology mapping and knowledge alignment techniques that integrate semantic data from different sources, as well as storage and search of semantic data. Within this middleware, our Semantic Desktop search engine contributes to these services:

Knowledge Structure Ontologies that are especially valuable for knowledge workers in the scientific domain are available (see Section 3.4.3).

Wrapping Pieces of semantic information, for instance, stored in files, e-mails, or on Web pages, are made accessible (extracted) to the Semantic Desktop (see Section 3.4.4).

Metadata Storage Extracted semantic data are centrally stored and integrated into one large semantic graph. Efficient search and browse facilities are provided to middleware services and semantic applications (see Section 3.4.5).

Mapping & Alignment Extracted semantic data are enriched and aligned to improve their quality and the benefit of the system (see Section 3.5).

Before I present our Semantic Desktop search engine in detail, I briefly describe the architecture of its base system, the *Beagle* Desktop search engine.

3.3.2 The Beagle Desktop Search Engine

Our Semantic Desktop Search engine is based on *Gnome Beagle*[10]. It was initiated in 2004 as an open source software project for Linux. It is designed to index and search over 250 different content types and supports a variety of data containers like the file system, e-mail and instant messaging applications, web browsers and address books. Data items such as files are indexed immediately upon creation or modification, e-mails and instant messages upon arrival, and web pages as they are browsed. Indexing is performed as a background service in an unobtrusive manner. *Beagle* uses the C# port of the Apache Lucene IR engine[11]. The *Beagle* core implementation is licensed under the MIT license[12].

At that time, *Beagle* experienced a lot of attention from various Linux Desktop systems. Though developed under the Gnome project, it is desktop-independent and was used by Gnome, KDE, and SUSE Linux Enterprise Desktop. It provides a graphical, a web-based, a command line, a programmatic library-based, and a network-based search interface to also query other *Beagle* instances over the network.

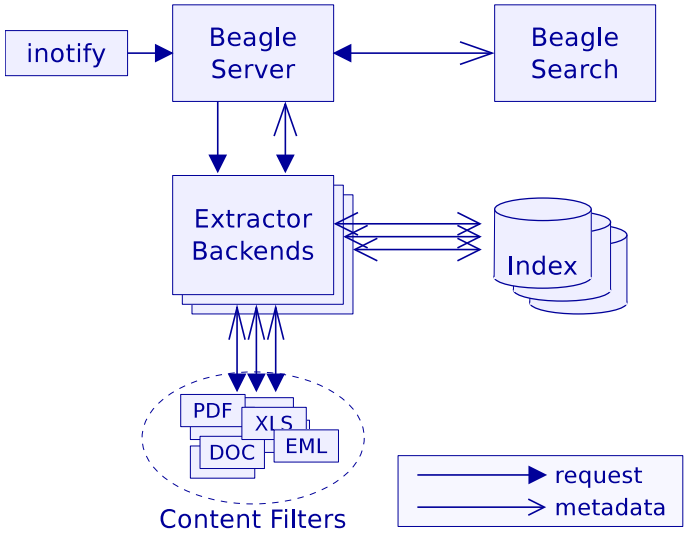


Figure 3.2: Architecture of the *Gnome Beagle* Desktop Search engine.

Figure 3.2 provides an overview of *Beagle*'s infrastructure. The *Beagle* server constitutes the core of the system. It initiates extractor backends, each responsible for a particular data container (*e. g.*, the file system or an e-mail application). The extractors pick Content Filters from a central pool to extract full-text and simple metadata. Each filter handles a particular set of content types. The extracted data are indexed in a Lucene index, one maintained by each extractor.

The *Beagle* search client is a stand-alone application providing a graphical search interface to the user. It connects to the *Beagle* server, sends a textual query entered by the user to the server

[10]Gnome Beagle: `http://beagle-project.org/`.
[11]Lucene.Net: `http://lucene.apache.org/lucene.net/`.
[12]OSI - The MIT License: `http://www.opensource.org/licenses/mit-license.php`.

and retrieves a list of results. The *Beagle* server parses the query, constructs a query model, and sends it to all extractors. They evaluate the query on their Lucene index in parallel and stream the results back to the server. The server merges the streams of results and forwards them to the querying client. As soon as the first results are retrieved, the client visualizes them.

The primary reason for choosing *Beagle* as our base system was, besides its open source license, the fact that it has a well designed modular architecture that allows for an easy integration of our extension. Further, the search functionality is accessible via command line which programmatically simplifies the automatic evaluation of such a system.

3.3.3 The Semantification of Beagle: Beagle^{++}

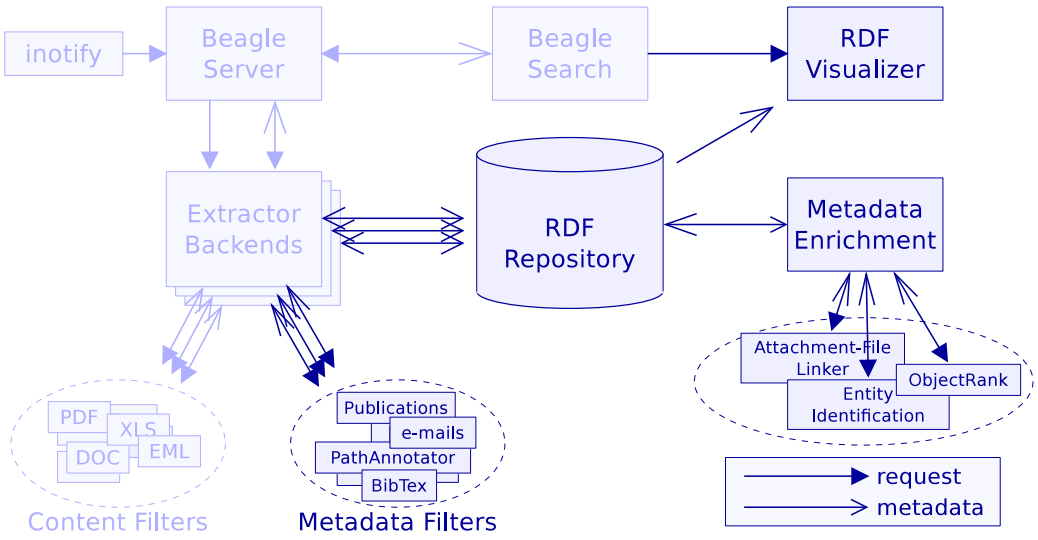
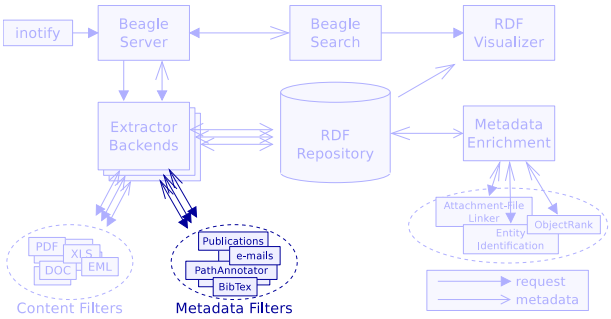


Figure 3.3: Architecture of the *L3S Beagle*$^{++}$ Semantic Desktop Search engine. The light-colored components refer to original *Beagle* components as shown in Figure 3.2.

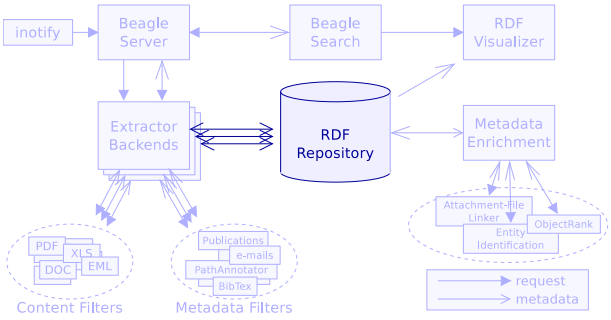
Starting from the original architecture shown in Figure 3.2, I extended *Beagle* towards our Semantic Desktop Search engine *Beagle*$^{++}$ depicted in Figure 3.3 by four major components. The indices of all extractor backends were integrated into one central RDF repository. These backends were further given access to a new pool of metadata filters that extract RDF fragments from data items such as files or e-mails. A number of metadata enrichment processes directly work on the RDF repository and create new knowledge. The search process was finally enhanced with a visualization tool for RDF data.

Modifications to the existing code were kept to a minimum in order to benefit from the high maturity of existing tested code, and to be able to also benefit from future bug-fixes of the original code. In the following, the four major components are described in further detail.

Metadata Filters: In extension to the existing *Content Filters*, our Metadata Filters extract complex semantic information from data items. These may include resources other than the processed data item, such as contained people or locations. Since these filters can create any kind of RDF metadata, they allow for generation of a rich network of knowledge. The architectural extension of *Beagle* includes a generic coupling of Metadata Filters with the *Beagle*$^{++}$ system. All our Metadata Filters are written in Java, whereas any programming language can potentially be used. The extracted metadata are passed to the *Beagle*$^{++}$ system by the use of standard formats and Inter Process Communication (IPC) mechanisms.

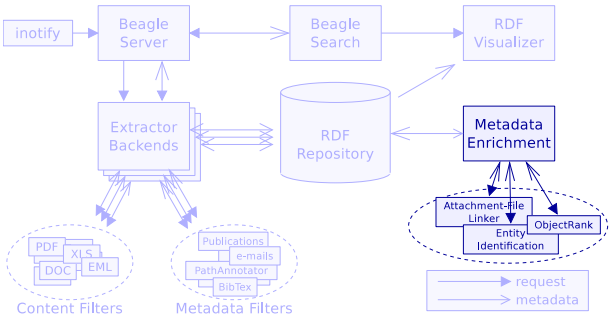


RDF Repository: The most visible modification done to the *Beagle* system was the integration of individual indices of the extractor backends into a central RDF repository. With this semantic store, all RDF data get integrated and accessible in a central location rather than being stored in application-specific silos and formats. The data can easily be accessed from external applications. This central store allows for semantification of applications and to make metadata available across application boundaries, thus opening information silos.

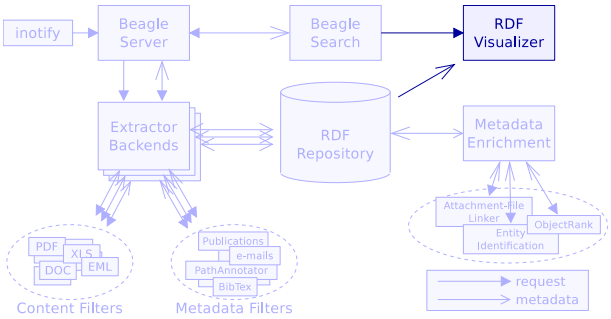


The primary design principle was to make this modification happen transparently to the extractor backends. For them, the central repository behaves as the individual indices did. For our system, in this repository all metadata integrates into one huge interconnected web of metadata. The two components Metadata Filters and the RDF Repository are discussed jointly under the topic of metadata generation and storage in the next section.

Metadata Enrichment: The central RDF repository allows additional applications to work directly on the extracted metadata. The Metadata Enrichment component allows for easy implementation of applications that consume metadata from the central RDF repository and create new knowledge. The coupling of this component is again done with open communication protocols and data formats. As for the Metadata Filters, this architecture allowed several members of L3S to implement and easily integrate their algorithms and applications into the *Beagle*$^{++}$ system to fill it with rich metadata extraction and intelligent services. In Section 3.5, I further describe these services.



RDF Visualizer: Another application that makes use of the direct access to the RDF repository is the RDF Visualizer. It is employed by the *Beagle* search interface to provide access to metadata of search results and to navigate from these through the entire metadata graph. This tool allows the user to learn and remember the context of search results and knowledge stored on her Desktop. Section 3.6 is particularly devoted to this subject.



The architecture presented in this chapter allows application developers to reuse our *Beagle*$^{++}$ system as a Semantic Infrastructure, *i. e.*, storage, integration, and search of RDF data. Therefore, extracted metadata no longer have to be isolated from other applications.

I will now go into detail on these four extension components throughout the following sections.

3.4 Metadata Generation and Storage — Implicit Knowledge Made Explicit

In order to have a system provide fast and efficient search capabilities to the user, it first has to process available user data. From all existing storage facilities like file systems, e-mail folders, address books, *etc.*, textual data (the full-text) and structural data (things and relations among them) are to be extracted. After extracting structural data, it is necessary to find means of articulating them in a machine processable way in order to store them persistently. Here, the first problem to tackle is the unique identification of things. Identical things observed at different locations should be referenced by the same identifier, in order to allow for instant integration of metadata. Further, the metadata[13] about a particular thing needs to be encoded using a certain vocabulary. Finally, the extracted and articulated textual and structural data are stored and indexed for quick search later on. I now elaborate on all of these issues in more detail.

3.4.1 Metadata Generation

The generation of structural data from the user Desktop can be achieved in two ways:

Explicit articulation by the user The user articulates each piece of information explicitly, *e. g.*, create a person, link e-mails to that person, maintain contact information, attach projects and tasks to them. This requires significant effort, but guarantees high quality knowledge. Further, the user develops her own schema of the data — a schema the user is an expert in. By this she will easier remember how knowledge is organized on her Semantic Desktop. The drawback is that such a personal schema requires ontology mapping techniques [178] when personal knowledge is to be published on the Web or exchanged within the social network of the user.

[13] We use *structural / structured data* and *metadata* synonymously.

Extraction of implicit structure The system identifies occurrences of entities and maintains the extracted metadata about them, *e. g.*, people are extracted from the address book, e-mails are linked to them based on the `TO:` and `CC:` header of the e-mail. This requires minimal effort for the user. It scales much better than the explicit strategy but results in a lower quality of extracted knowledge. Further, the system uses its own schema for organizing the knowledge, which might be different to individual users' way of organization. Therefore, the user first has to learn the system schema and adapt to it.

In our prototypical study, we focused on the automatic extraction of implicit structure to allow for a low burden for the user to start using our system. However, within the context of the NEPOMUK project, we also had the opportunity to study explicit knowledge articulation methods [114, p. 17–23] and test our further achievements on higher quality knowledge having a higher diversity of schemata (see Section 3.6 and Chapter 4).

3.4.2 Unique Resource Identification

The automatic extraction of implicit knowledge requires a way to globally uniquely identify entities. This ensures that two metadata fragments about different things generated at different points in time do not unintentionally integrate into one metadata graph. For this, a unique identifier (uid) of one distinct entity must not be used for any other entity.

An optimal solution would pick the same id for the identical entity at different occurrences. E-mails, for instance, provide a globally unique identifier in the `Message-id` header attribute. Whenever this e-mail occurs in the header of another e-mail, this uid can be reused. Then, all metadata about this e-mail (including all referring e-mails) integrate into one connected graph. E-mail addresses or Web page URLs are other such examples.

If available information for an occurring entity is insufficient for unique identification (for instance people extracted from texts where only names occur), an identifier has to be used that cannot be generated for any other occurrence of *any* entity. To stay with the example, an identifier composed of the uid of the text document and the person name could be used. Then, at a later step, identifiers that actually refer to the same entity have to be identified in the global graph and connected or merged. We elaborate on this second phase of metadata generation in Section 3.5.

3.4.3 PIM Ontologies

A modular metadata extraction architecture has to use a common ontology. Otherwise, extracted metadata fragments cannot integrate into one large connected metadata graph. Then, mappings between the different ontologies are needed, which requires a significant computational effort.

On the Desktop, some file types like JPEG images contain explicit metadata for which consequently type-specific ontologies exist (*e. g.*, EXIF metadata[14]). Additionally, Desktop applications have their own domain and application-specific requirements on metadata. An ontological

[14]EXIF: `http://www.exif.org/specifications.html`.

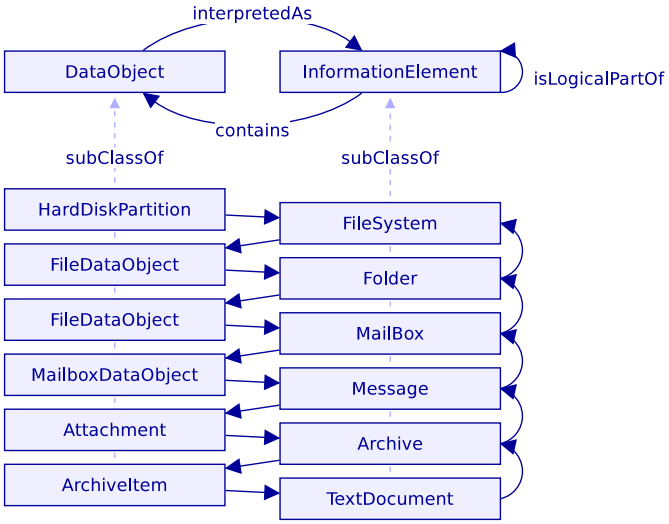


Figure 3.4: Example of recursive Data Object and Information Element relationship.

solution should address both, resource-specific *and* application-specific requirements. Such a solution is provided by the NEPOMUK ontologies[15] [150, Ch. 5] comprising these main parts:

NEPOMUK Representational Language (NRL): NRL [173] bases on RDFS, whereas all RDFS schemata are *legal* NRL, but only those schemata that follow NRL usage recommendations are *valid* NRL. NRL add the notion of named graphs to RDFS, as well as graph views. This allows for making RDF statements about RDF statements, without introducing the problems of reification (`RDF:Statement`).

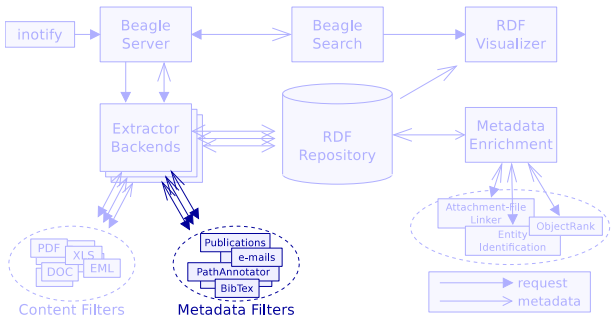
NEPOMUK Information Elements framework (NIE): The NIE ontology makes the particular distinction between *Information Elements* (IE) like e-mails or a file, and the actual *Data Object* (DO) encoding it. A Data Object is interpreted as an IE, whereas an IE can in turn contain another Data Object. Figure 3.4 depicts an example of this recursive relation between Data Objects and Information Elements. There, a hard disk partition (DO) can be interpreted as a file system (IE), whereas a file system contains file Data Objects (DO), which in return can be interpreted as folders (IE) containing other file Data Objects, being interpreted as an e-mail mail box or files, and so on. The distinction between Data Objects and Information Elements allows the latter to be stored in different Data Objects, *e. g.*, as caused by saving e-mail attachments to disk.

The NEPOMUK Ontologies provides us with a vocabulary for a broad range of Personal Information Management (PIM) applications. Further, it is designed to be extended to fit a specific domain, while staying compatible with other applications that use NEPOMUK Ontologies. Therefore, it perfectly fits the needs of a Semantic Desktop Search application. We use the NEPOMUK Ontologies to design two domain-specific ontologies on top of it. The Publications

[15]NEPOMUK Ontologies: `http://nepomuk.semanticdesktop.org/ontologies/`.

Ontology models scientific publications[16], which is primarily used by our Scientific Publications extraction module. The second ontology contains a vocabulary that is specific to our prototype implementation[17] and is primarily used internally.

3.4.4 Metadata Extraction Process



On Desktop PCs, content and metadata are stored in a multitude of files and other data containers, using a particular content type[18]. Each of these content types has to be processed in a specific way to access content and metadata. The same content type can be reused in different locations, for instance, the `text/plain` content type can be found in simple .txt files, in e-mail messages or attachments. The `image/jpeg` content type can be found in .jpg and .jpeg files, as well as in e-mail attachments. Therefore, the extraction process can reuse content type specific implementations for different containers (files, e-mail, zipped archives).

Existing Desktop Search tools (see Section 3.2) extract flat metadata as attribute-value pairs for files, e-mails and other Desktop containers. Such attribute-value pairs provide, for instance, the name or size of a file, the author or title of a document or e-mail, and so on. These can easily be transformed into structural data, where the entity these tuples refer to is the subject, the attributes are properties, and values are literals. These metadata are here called flat, since they do not link to other resources. A Semantic Desktop Search tool in addition extracts relations among entities, *i. e.*, file-folder structure, e-mail attachment-store-as-file links, e-mail reply-to relations as well as authorship links to the actual person entities (and not only persons' names).

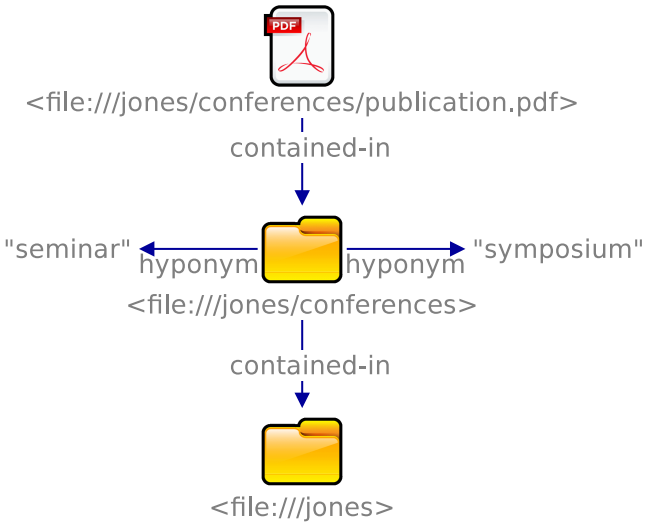
A primary design principle of the metadata extraction phase is, in contrast to the metadata enrichment phase described later on, that for extraction only the data item being currently processed is used in order to generate metadata. In this phase, there is no read access to the RDF metadata that was previously generated and is now stored in the repository. Any metadata generation that requires existing knowledge from the repository is postponed to the metadata enrichment phase. This decision improves performance, which is important at this phase, and simplifies the architecture. The Metadata Filters are light-weight applications that are given the data item to process. They do not need to implement means to access the repository, but simply stream their metadata to the *Beagle*$^{++}$ system via IPC methods.

For our particular Semantic Desktop Search prototype implementation, we created the following metadata extraction modules. These extract complex metadata as soon as the Content Filters have finished. This allows to integrate the output of both types of filters, and to nicely reuse the large number of existing Content Filters.

[16]Publications Ontology: `http://beagle2.kbs.uni-hannover.de/ontology/publications`.
[17]L3S Ontology: `http://beagle2.kbs.uni-hannover.de/ontology/l3s`.
[18]The Internet Assigned Numbers Authority (IANA) lists around 1,200 distinct content types under `http://www.iana.org/assignments/media-types/` (15.07.2010)

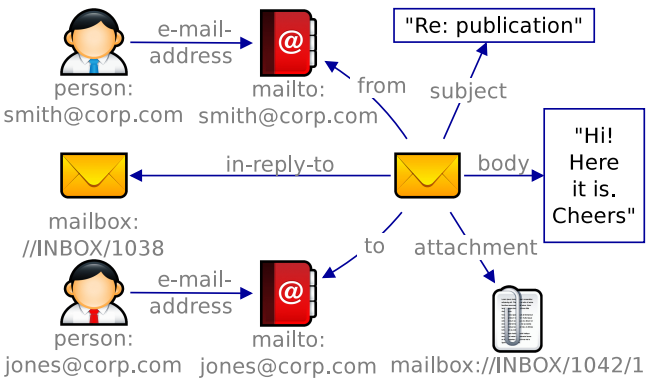
Web Cache. The cache of Web browsers store web pages visited by the user. Extracting the link structure of these pages allows for better ranking of web pages after keyword search, as the well-known PageRank algorithm has shown [141]. Further, visited links are especially highlighted in the Web browser so that the user easily remembers past trails of recent navigation.

File Paths. Users invest a significant amount of cognitive effort in organizing files into a hierarchical classification. However, current Desktop Search systems hardly take this structure into account. This classification, however, has some drawbacks. The user has to classify files when she stores them, anticipating the situations when they might to be remembered [105, p. 24]. For example, pictures taken in Hannover might arguably be stored in a folder called "Hannover" or "Germany". However, later, the user might only remember one of the two keywords, depending on her context, which would render the query to fail finding the images if the wrong keyword is used, though both have a clear semantic relation.

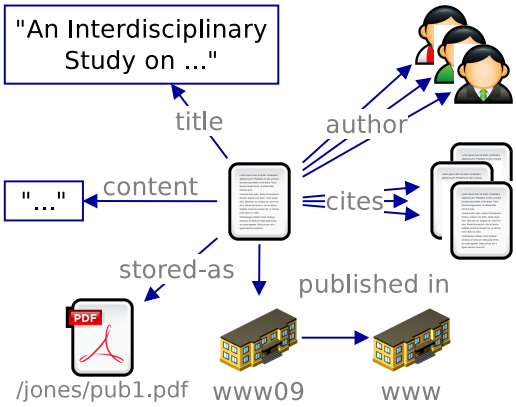


The PathAnnotator extractor annotates files with terms occurring in their file path. Therefore, an image stored in the folder "Hannover" can be found with that keyword, though it does not appear in the image file name itself. Further, each such term is expanded by semantically relevant terms like synonyms, hyponyms or hypernyms, as provided by WordNet [119]. This allows for finding images stored in the "Hannover" folder by the keyword "Germany", simply because Hannover is located in Germany.

E-mails. E-mails provide a lot of structured information about people, their e-mail addresses, and their communication with other people. An e-mail sent as a reply to another e-mail refers to that other e-mail. With such structured data at hand, queries can be articulated that restrict keywords to appear in e-mails that were sent (maybe as a reply) to a particular person. The person herself can be defined by keywords or even structural information like affiliation. As e-mails are processed by our extractor, a network of people, their e-mail addresses, e-mails sent, received and replied to, as well as corresponding attachments is created.



Scientific Publications. In the research community, publications such as conference or journal articles represent the main information source. These publications form a network of authors, conferences and related work via co-authorship, published-at, and citation relations, respectively. Today, such publications can most easily and quickly be found on and retrieved from the Web.

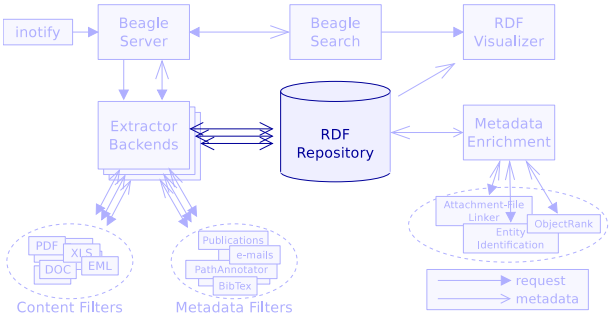


During the process of developing a Ph.D. thesis, one downloads and successively skims, reads or thoroughly studies thousands of such publications. Remembering that something relevant was read before is easier than refinding the actual publication. Using the network between publications matching a query, similarly to the PageRank algorithm [141], relevant publications, authors, or conferences can be found by the system. Further, assuming only publications relevant to the user are downloaded, conferences and authors that those publications frequently link to are more relevant than others.

The Scientific Publications module extracts the above mentioned metadata from publications, which integrate into a large network. This module uses data from the DBLP database on Computer Science publications[19], hosted and enhanced at our institute towards Faceted DBLP[20].

3.4.5 Metadata Storage and Indexing

Metadata generated in the previous phase are immediately stored in the central RDF repository. Due to the use of ontologies and the resource identification strategy discussed in Section 3.4, the graph fragments of each filter integrate into one large interconnected graph. For the full integrated metadata graph see Figure 3.6 on page 48. There, the example metadata fragments shown in Sections 3.4.4 and 3.5 are shown in one figure. Each resource that is part of multiple metadata fragments integrates these fragments into one RDF graph.



In *Beagle*'s infrastructure, all extractors store their extracted metadata in individual indices. This architecture clearly follows the information silo paradigm introduced in Section 2.2, which we aim to overcome while keeping as much original code untouched as possible. Our system firstly transforms the extracted metadata originally encoded in Lucene documents into RDF graphs to be stored in the central repository. This transformation reuses the unique identifier of the resource the metadata are extracted from as the `subject` URI of all generated RDF statements. For each `<attribute,value>` pair of the metadata, a statement is created where the `predicate` reflects the attribute, and `object` the value. This transparent transformation is depicted in Figure 3.5.

[19]The DBLP citation database: `http://dblp.uni-trier.de/`
[20]Faceted DBLP: `http://dblp.l3s.de/`.

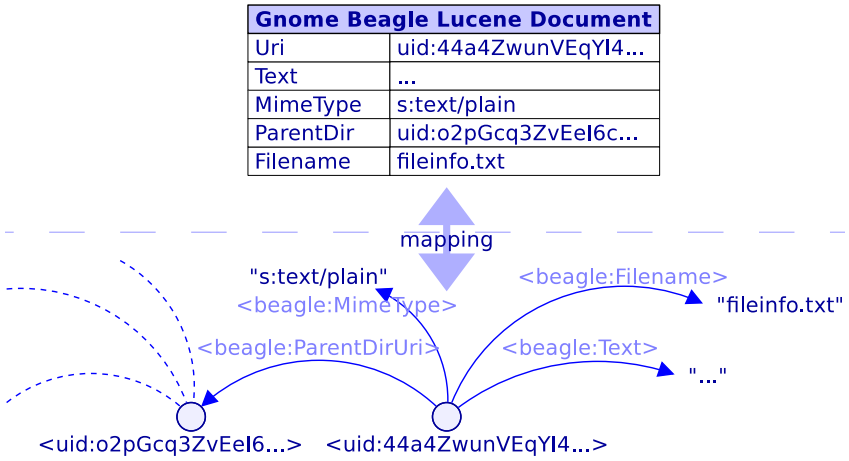


Figure 3.5: Transparent mapping between an IR (Lucene) and RDF model (graphs).

The centralized store is transparently integrated into *Beagle*'s infrastructure, *i. e.*, the store mimics the individual indices of the extractors. With this approach, we can reuse the extractors without any modification. Since the extractors perform particular actions during search, we can also reuse those implemented actions. The file system extractor, for instance, transforms an extractor-specific id of files into actual paths. Such activities require significant integration effort when migrated into a single search component. In our approach, this tested code can be reused.

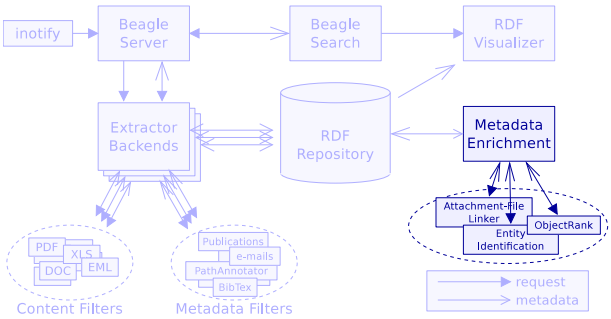
However, from an architectural point of view, a single search component working on the central repository would certainly be a better solution. We therefore see such a central search component as a potential performance improvement and our system's performance as an upper bound for achievable performance, which is an acceptable property of a prototypical implementation.

Finally, existing extractors can only find resources that Content Filters extracted. These are, for example, files for the file system extractor or e-mails in case of the e-mail extractor. Even though they also employ Metadata Filters to extract complex RDF graphs, they are not aware of these metadata. Examples of such RDF graphs are resources as people or conferences extracted from files or e-mail attachments. Again, significant modification of the original extractors would have been necessary. Instead, for all those new resources we further added an extractor module that provides the search client with all those new resources matching the user's query.

As our central data store, we use the NEPOMUK RDF Repository[21] [123, Sec. 5.1] that I developed in cooperation with other partners within the NEPOMUK project [131]. It is a Java-based, stand-alone server accessible via the OSGi[22] and SOAP[23]. Once the RDF data are added to the central store, it provides means for quick access to the data using standard query languages. For further details on the used implementation, please refer to Chapter 5 on page 69.

[21]The RDF Repository of the NEPOMUK project: `http://dev.nepomuk.semanticdesktop.org/wiki/RdfRepository`.
[22]OSGi Alliance: `http://www.osgi.org/`.
[23]SOAP Specification: `http://www.w3.org/TR/soap/`.

3.5 Metadata Enrichment — Creating New Knowledge



As we saw, during the metadata generation phase, small fragments of metadata are extracted from data containers. At that stage, only knowledge that is locally available in the isolated context of the particular container is used for performance reasons. Any activity that requires knowledge beyond that scope is postponed to the subsequent phase of metadata enrichment.

Now at this phase, having the global knowledge at hand, more knowledge can be derived from the composition of the single fragments. During the previous section, we already mentioned some possible sources of additional knowledge. We created four metadata enrichment modules: Firstly, we use the ontologies defined earlier to entail new metadata that is implied by the existing knowledge. Secondly, we apply domain-specific heuristics to reconstruct saved-to links from e-mail attachments to files. Thirdly, we aim at identifying resources that have distinct ids that actually refer to the same entity. Finally, resources are scored with a rank representing their global importance within the network. These precomputed global scores are later used to improve ranking during search.

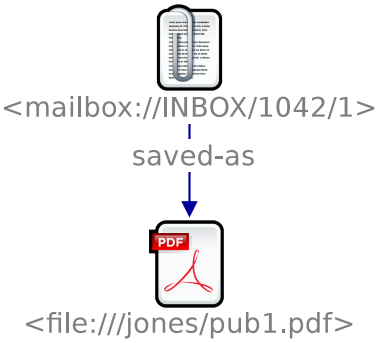
All these enrichment modules are executed asynchronously from the extraction module. Its execution can be triggered following two strategies. Firstly, it can be run in a periodic manner. Alternatively, it can be executed when significant changes are done to the data store. Since the latter requires a notification mechanism that is not available in mature data stores, and the implementation of that feature is out of scope of our work, we opt for the former approach.

Reasoning for new Knowledge Semantic data stores allow for reasoning new knowledge from existing one. Fundamental logic rules, the axioms, are used to make logically valid knowledge explicit that is implicitly given by the data. Such axioms exist in RDFS and OWL. The former allows for sub-class and sub-properties relations only, where reasoning provides the transitive closure of these hierarchies. The latter allows, among others, for defining relations of your own ontology being transitive, (a-)symmetric, (ir-)reflexive, and (inverse-) functional (see Section 2.1 and [133, Sec. 2.3.2]). The more inferencing capabilities of OWL come with higher computational effort which are sometimes not guaranteed to come to conclusions [132, Sec. 5].

The reasoning process can be performed in two different phases in the data store. It is either performed while adding new data to the store, which are then entailed so that the store contains the explicit and implicit knowledge. Alternatively, the inferencing is performed while querying the explicit knowledge only. The former reduces the indexing performance measured in indexed statements per second, but increases the storage requirements. The latter omits indexing overhead and preserves storage capacity but reduces performance of query evaluation.

In the work done within the scope of this thesis, we focused on RDFS inferencing at indexing time, providing us a limited amount of reasoning, but with acceptable computational overhead.

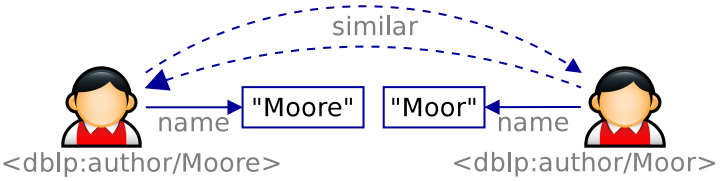
Attachment Linking With the act of storing an e-mail attachment to disk as a file, its rich context — the sender, the e-mail message that refers to the attachment, the prior e-mail asking for that file — is lost. These semantic links are of value for the user [181], but she has to make herself aware of them later on.



Our attachment linking module reconnects the e-mail attachment with the file on the disk, to allow for exploiting these semantics for searching and browsing. For this, the module searches the data store for files and e-mail attachments that have the same file size and file extension, with the file's date of creation being after the arrival time of the e-mail. The file names of the file and the attachment are compared for string similarity[24]. This is based on the usual behavior to change the file name only slightly when stored to disk.

A more reliable solution would certainly be an extension of the e-mail application to explicitly create that semantic link when the user performs the action of storing an attachment as a file. For the sake of compatibility with different e-mail clients, we decided for the presented heuristics. However, as we discussed in the Chapter 2, it is desired to open up the information silos of classical monolithic applications to participate in a Desktop wide semantic infrastructure, since this would much more easily create high quality semantic data.

Entity Reconciliation Our e-mail and publication filters generate entities referring to people. The former uses the label of the e-mail address, the latter uses authorship information of publications. Due to multiple e-mail addresses used by the same person, or different ways of writing a person's name (including typing errors), we end up to have multiple entities in the data store that actually refer to the same person.



To exemplify such a situation, lets assume a user stores two publications from "Moore" *et al.* and "Moor" *et al.*, whereas the first author is actually the same one that is once mis-spelled. The co-authorship and reference network of the publications may provide sufficient evidence to conclude that those two instances actually refer to the very same person.

The task of Entity Reconciliation (also called Reference Reconciliation [189]) is to identify those resources that are very likely to be identical. The decision of a resource to be identical to another is based on the similarity of the literals of both resources, and the similarity of referenced resources. This hypothesis is tested using a Bayesian Network of beliefs [101].

Global ObjectRank Web search engines have proven that the query relevance of documents yields better ranking when combined with structural properties of the documents [12, Sec 13.4.4]. In order to have our Semantic Desktop Search system to provide a better ranking, we employ a similar technique to encode connectivity of resources in a score and combine this with the traditional ranking of full-text search, as we show in the following.

[24]The SecondString library: http://secondstring.sourceforge.net/

On the Web, the link structure between documents is used as a voting system [111, 141]. A Web page which many pages link to has a large sum of votes. The weight of each page's vote is reciprocal to the number of its outgoing links, and is proportional to the sum of votes it has itself.

A similar but in a sense two-phased voting system is the HITS approach [108, 111]. Here, two weights are computed that mutually reinforce. One is the authority weight of a page that is the sum of the hub weights of all pages that point to it. The hub weight is in return the sum of the authority weights of the pages pointed to. An authority Web page for a specific query is very likely to provide relevant information *w. r. t.* the query, whereas a hub Web page provides links to good authorities. A fundamental property of HITS is that it is not computed on the entire Web graph but only on a sub-graph surrounding the result of a user query. These pages are relevant for the topic of the query. HITS then finds hub and authority pages in that sub-graph. We integrate a similar PageRank-based approach into Desktop Search in Section 3.6.1.

A different interpretation of the PageRank scheme is the Random Surfer Model [141], which bases on a simplistic model of a Web surfer visiting Web pages. This model assumes that a Web surfer follows links of a Web page with even probability or performs a random jump (teleportation) to any other Web page on the Web. In this Markov chain [111], the probability that a random surfer is located at a given page at any time is proportional to the probabilities that the surfer was located at pages linking to this page in the previous step. In other words, the score of a page is proportional to the sum of the score of all pages linking to it.

This method provides an importance score for each Web page based on its connectivity. Similarly, ObjectRank [13] provides an importance score for relational data. In contrast to Web pages, relational data have typed links where different types may transfer votes with different weight, here called authority. ObjectRank defines an *Authority Transfer Schema Graph* that extends the schema by link type weights to control how importance propagates along the structure.

A connectivity matrix $\mathbf{A} = (\mathbf{a}_{j,i})$ is a column-stochastic matrix generated from the actual data where each entry $a_{j,i}$ holds the weights of the links from resource i to j. Then, the PageRank formula is applied where authority r_i of resource i is computed as the sum of all resources' authority that are connected to it, weighted by the connection weight $a_{j,i}$. Having $\mathbf{r}$ being the vector of resources and $\mathbf{A}$ the adjacency matrix, this recursive relationship is the solution of the linear system of equations

$$\mathbf{r} = d \cdot \mathbf{A} \cdot \mathbf{r} + (1 - d) \cdot \mathbf{e} \tag{3.1}$$

The solution $\mathbf{r}$ is computed using the power method [111, Sec. 5.1]: $\mathbf{r}_0$ is set to $\mathbf{I} = (1, \ldots, 1)^T$, and $\mathbf{r}_{k+1} : k \in N$ is computed iteratively with

$$\mathbf{r}_{k+1} = d \cdot \mathbf{A} \cdot \mathbf{r}_k + (1 - d) \cdot \mathbf{e} \tag{3.2}$$

where usually $d = 0.85$ [111], until $\mathbf{r}_k$ and $\mathbf{r}_{k+1}$ converge, *i. e.*, $|\mathbf{r}_{k+1} - \mathbf{r}_k| < \varepsilon$.

To apply this method to our system, we annotated our ontology with an authority transfer schema. Then, a structural importance score is available for each resource, which we use to improve relevance score during search. The ObjectRank only needs to be computed on significant metadata changes, which usually happen less frequent than search is performed.

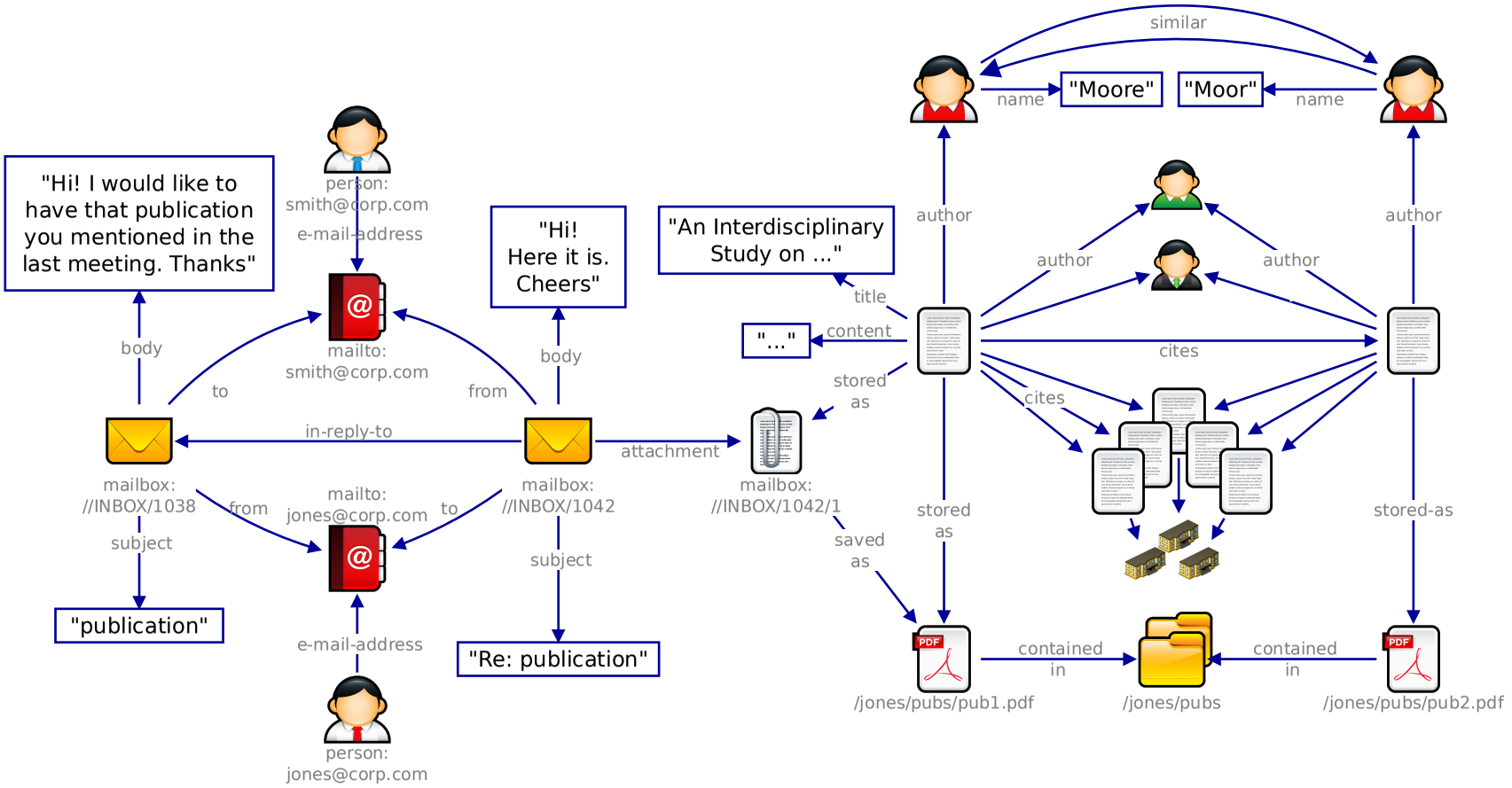


Figure 3.6: Example metadata graph as it integrates the metadata fragments of different metadata extractors and enrichment modules: a publication and its citation network (middle right) linked to a file it is stored in (middle, both extracted by the publications filter), the file system structure (bottom right, the paths extractor), two e-mails, one with the PDF file as its attachment (left, e-mail filter), stored to disk (middle bottom, attachment linker), and two distinct author resources linked as similar since they actually refer to the same person (top right, entity id).

3.6 Metadata Search

Now that we have set up a semantic infrastructure that extracts and indexes user's data in a structured way, we can provide the user with search that exploits this structure. The user can express the structure or semantics of her information need in two principle ways:

Implicit Semantics. The user expresses search criteria implicitly, using keywords or natural language. The search system interprets this unstructured query and identifies the intended meaning. Usually, all possible structured interpretations constitute to the query result.

Explicit Semantics. The user explicitly states the semantics of her query, thus reduces the ambiguity of the query and increases the relevance of matching results *w. r. t.* her intent.

To study the benefits of Semantic Desktop Search, we opt for explicit semantics to give a user full control over her query's meaning, while enabling us to measure benefits introduced solely by semantics. Therefore, we extended the keyword search by explicit semantics. Keywords can be bound to particular properties like the publication title or e-mail body. By doing this, the type of resources the user is searching for also gets restricted. To reduce the burden of typing lengthy predicates to link keywords to, we employ mappings from special terms to their URI representations. For instance, the term `"author"` maps to the authorship property of a publication `desktop:desktop_document_author`[25], and `"name"` maps to name property of a person `desktop:person_name`. Furthermore, properties can be aligned to paths, such that keywords can match in a defined distance from the actual resource being searched for. As an example, the keyword `"author:name:john"` only matches publications that have an author whose name matches `"john"`.

Such a query can easily be translated into a structured query in SPARQL. Please refer to Chapter 5 on page 69 for details on notation of keyword search in SPARQL and its efficient evaluation. In Chapter 6 on page 103, I investigate the performance of keyword search of competitive semantic data stores. In a next step, retrieved results are processed to improve their rankings and include results that are similar to the matching ones. Finally, results are visualized to the user.

3.6.1 Incorporating ObjectRank into Ranking

Results matching a keyword query get a textual result-to-query similarity-based score. In our case, this is the TF×IDF score provided by Lucene. However, this does not take into account the connectivity of the results within the whole metadata graph. This context is of particular value to the user, as it has been shown in [181]. The context where certain things occur helps users to remember and find them. For this, we combine the TF×IDF value with the ObjectRank R:

$$\mathrm{R}'(r) = \mathrm{R}(r) \cdot \mathrm{TFxIDF}(r), \tag{3.3}$$

where r is the respective result. Then, only those results with a high relevance to the user query (high TF×IDF value) *and* high connectivity in the metadata graph get a high overall score.

[25] Prefix `desktop:` refers to `http://beagle2.kbs.uni-hannover.de/ontology/desktop#`.

Local ObjectRank

The ObjectRank computation given in Equation 3.1 further allows for query-dependent scores. The $\mathbf{e}$ vector models the probability of the random surfer to teleport to specific resources. When only one resource is given, the result ObjectRank score reflects how authority propagates from the resource through the network, or in other words, how important it is to all other resources.

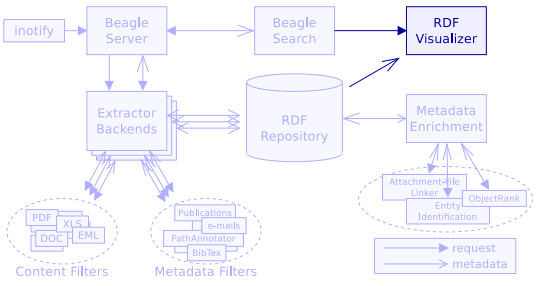
Given a set of resources to teleport to, the resulting scores are the linear combination of importance scores of all resources in that set [92]. We use the full-text search scores of matching resources as the $\mathbf{e}$ vector to compute a local, query-dependent ObjectRank. Similar to the Random Surfer Model, this models the probability that the user visits a particular resource not only by search but also by navigation from any existing resource. Nodes (*e. g.*, a conference or an author) that are connected to multiple search results (*e. g.*, publications) may gain a larger score than the respective search results as they accumulate propagated authority from multiple results. This behavior allows to identify resources that are relevant for certain search results but are not part of the search result set themselves.

3.6.2 Extension by Similar Results

In order to also provide results that actually did not match the query, but are very similar to matching results, we incorporate the similarity information derived by our entity reconciliation module. This, for instance, provides results that contain a misspelled name, where the correct name was used as a keyword. For this, the semantic store is queried for all similar results of the original result set. The score of those similar entities $r(s)$ is computed based on the score of the respective matching result $r(m)$ and the similarity $sim(m, s)$ between the matching result m and the similar result s:

$$r(s) = r(m) \cdot sim(m, s) \tag{3.4}$$

3.6.3 Visualization of Search Results



The main challenge in visualizing search results is to present a large number of results in a clear and well-arranged way. The user must not be overwhelmed with the full amount of available information. The purpose of search result visualization is to provide the user with an overview of results that allows for a quick grasp. Therefore, only a very limited amount of information can be presented for each result (see Figure 3.7). Usually, these comprise an icon or preview image, a very short textual value uniquely identifying the result, and a longer more descriptive textual value.

Numerous systems and interfaces that deal with finding and displaying Desktop resources exist. Their approaches can be categorized along two orthogonal dimensions: static *vs.* inter-

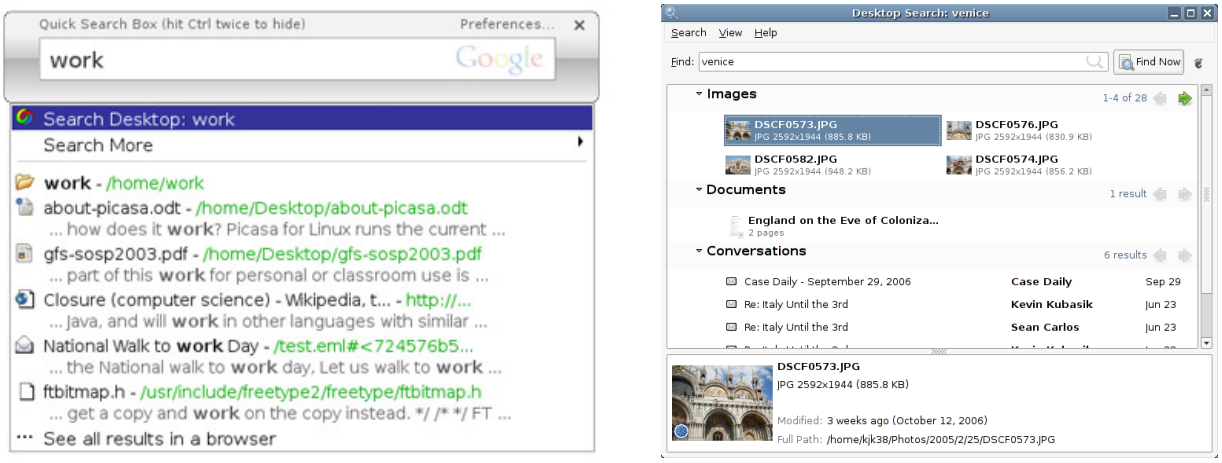


Figure 3.7: Search result visualization by *Google Desktop Search*™ and *Gnome Beagle* .

active[26] [144] and graph-centric[27] [4] *vs.* node-centric[28,29]. The first classification refers to the output of the visualization tool; whether it is fixed or can be manipulated by the user. The latter refers to the proportion of the displayed graph; whether it is the entire graph or only a fragment.

HP Labs developed *An Experimental RDF Graph Visualizer* which shows the surrounding sub-graph of a resource in a static node-centric way, but provides browsing and full-text search. The visualizer is running as a servlet using Scalable Vector Graphics (SVG) and is therefore platform independent and easy to integrate into other applications. However, on startup, it first has to read the entire graph into the main memory which prevents it from being used for large graphs. *Beagle*$^{++}$, which uses an enhanced version of this tool integrated into the *Beagle* Search interface, does not face this problem, since we always visualize smaller metadata sub-graphs. Other systems like OntoRama [72], RDFSViz[30], OntoViz[31] are specifically designed for visualization of ontologies, which is too restricted compared to our purposes.

Depending on the type of result, different pieces of information are to be used for visualization. For an e-mail, for example, the subject, the sender, and reception date are candidates, whereas a PDF or word processor file provides title, last modification date, and preview thumbnail.

The visualization of Semantic Desktop Search results in contrast turns out to be more difficult. The generic nature of the RDF framework allows for an arbitrary number of different result types. From an implementation point of view, this cannot be handled in a hard-coded fashion anymore, *i. e.*, by defining the visualization for each result type. The visualization rather has to be as generic as its data. We therefore decided for a solution where the search results themselves define how they are to be visualized best.

[26] RDF Gravity: `http://semweb.salzburgresearch.at/apps/rdf-gravity/`.
[27] SIMILIE Welkin: `http://simile.mit.edu/welkin/`.
[28] Fenfire: `http://fenfire.org/`.
[29] Fentwine — A navigational RDF browser and editor: `http://www.w3.org/2001/sw/Europe/events/foaf-galway/papers/pp/fentwine/`.
[30] RDFSViz: `http://www.dfki.uni-kl.de/frodo/RDFSViz`.
[31] OntoViz: `http://protege.cim3.net/cgi-bin/wiki.pl?OntoViz`.

The extraction modules are responsible for annotating those metadata of extracted resources that are useful for visualization. For this we use the `rdfs:label` and `rdfs:comment` properties as the only textual metadata for visualization. Either, these properties are directly generated by the extraction modules, or the ontology used by the modules defines which properties are to be interpreted as `rdfs:label` and `rdfs:comment` via the sub-property annotation. Then, the inference step generates the required properties used for visualization.

Further, generic visualization information like a type-specific icon can directly be defined in the ontology which renders an annotation of each individual result unnecessary. The Fresnel vocabulary[32] allows for more complex visualization annotations than our prototype requires.

All classical Desktop Search systems provide the "open the result" methodology. With this, a particular search result can be opened with the appropriate application to further inspect or use the result. In contrast, the Semantic Desktop provides new kinds of entities that cannot be accessed or "opened" in the same sense. For instance, a scientific conference could be found by the search application, which occurs in multiple publications stored on the Desktop of the user. These rich metadata have to be made available to the user. Providing the user with an equivalent action would make such new resources accessible and more tangible to the user.

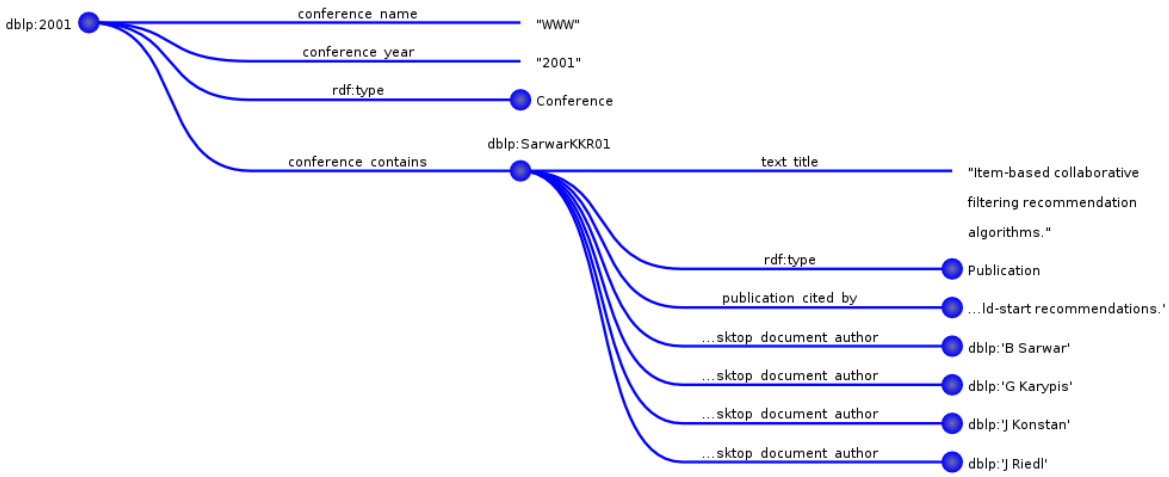


Figure 3.8: RDF Metadata visualization of search results.

We faced this challenge with an RDF graph visualization tool which is used to "open" any RDF resource. It bases on the *HP Experimental RDF Graph Visualizer* [167], extended by features and significant performance improvements. It visualizes the surrounding RDF metadata graph and allows for arbitrary browsing. With that at hand, the user can explore the meaning of results within the large network of metadata. Figure 3.8 depicts such a metadata graph visualization. Circles denote resources. The left-most resource is the one being "opened". It is connected to other resources and textual values, illustrated by lines. By clicking on another resource, that one becomes the left-most node, so that its surrounding network of metadata is shown.

[32]Fresnel Vocabulary: `http://simile.mit.edu/wiki/Fresnel`.

3.7 Experiments

For the evaluation of our Semantic Desktop Search engine *Beagle*$^{++}$, we compare its search results to those of its baseline system, the *Gnome Beagle Desktop Search* engine. With this comparison we directly see the enhancements introduced by semantifying an existing Desktop Search infrastructure.

We employ a user study where human judges rate the relevance of results provided by the evaluated systems. In a second experiment, we measure the performance of indexing and querying user data.

3.7.1 Data Set Description

For such an evaluation, a data set is required that exposes realistic characteristics of user Desktop data. Such data are highly personal, which poses two challenges:

Privacy. The data stored on a user's Desktop underlie certain privacy concerns. These concerns do not only regard the user herself, but also everyone else who contributed to the user's Desktop data like colleagues sending e-mails and documents to the user. Usually, users hesitate to publish such private data.

Personalization. User's data are highly personalized. It comprises a dense collection of data highly relevant to the interests and expertise of the user. A different user might comprise completely orthogonal data. Targeting at the user task of searching Desktop data, only the user who owns the data can query them with meaningful and realistic queries.

Primarily, due to the privacy issue, a publicly available Desktop data collection is not available and difficult to create and publish. Alternatively, a synthetic data set and queries could be generated. However, such an artificial setup would introduce additional implications on the quality performance of our evaluation setup, which we want to avoid.

We opt for the following strategy. Colleagues kindly devote themselves as volunteers and contribute Desktop data, as well as queries that reflect their everyday search, which therefore match their personal data. The users are allowed to pick the level of privacy of their data that

User	E-mails	Publications	Contacts	User	E-mails	Publications	Contacts
#1	30,627	0	0	#7	218	95	0
#2	4,423	869	0	#8	0	236	31
#3	833	236	0	#9	1,034	31	52
#4	3,820	266	10	#10	1,068	157	82
#5	2,012	112	0	#11	1,167	426	0
#6	217	28	0	#12	48	452	0

Table 3.1: Resources as contributed by users to the L3S Desktop Data Collection. In total, there are 45,467 e-mails, 2,908 publications and 175 contact information.

they were willing to contribute. All participants sign an agreement that only the participants get access to the data, so that they have a particular advantage in participating in this data set. Further, no data is allowed to be read by humans, except for debugging purposes. Therefore, major parts of the collection are only processed in an automatic way, as, for instance, done by our evaluation. Based on this agreement, twelve colleagues contribute personal Desktop data, as summarized in Table 3.1. The data collection comprises 48,550 items, consuming 8.1 GB of data [47, 48]. In addition to the data set, users further provide appropriate queries. We specifically ask for two queries for each of these types:

Clear Queries. Pure keyword queries that have a clear meaning, *e. g.*, `"desktop search"`.

Ambiguous Queries. Pure keyword queries that have multiple meanings, *e. g.*, `"beagle"`.

Structural Queries. Pure structural queries, *e. g.*, *e-mails sent by <minack@L3S.de>*.

Hybrid Queries Queries containing keywords and structure, *e. g.*, *e-mails sent by minack@L3S.de containing `"desktop search"` in the e-mail body*.

3.7.2 System Quality and Performance Evaluation

In our experiment, we evaluate three systems with these 96 queries: *Beagle* , *Beagle*$^{++}$, and *Beagle*$^{++}$ *with ObjectRank*. The difference between the latter two is that one does not employ the ObjectRank scheme whereas the other uses ObjectRank scores to re-rank search results. The users are asked to distinguish the top five results of their personal queries as being relevant or irrelevant. In Table 3.2 and Table 3.3, average values for precision at top-k (P@k) are given, *i. e.*, the precision among the top-k results. This is the number of relevant results in top-k divided by k. We measured $k = [1 \dots 5]$. Note that one outlier is removed from the analysis of the results, because the subject deviates by more than 70% from the average. Post-experiment investigations showed that the user rated the relevance of some results incorrectly.

In the second phase of our experiment, we target at the indexing and querying performance of *Beagle*$^{++}$ and compare it to *Beagle*. Here, we measure the time that the system requires to index the data set of a user. Further, we consider the querying performance to be the time from submitting a query to the system until the last result is retrieved.

3.7.3 Results and Analysis

Our user evaluation show that *Beagle*$^{++}$ outperforms the original *Beagle* system in all cases. This can be explained twofold. Firstly, metadata are primarily extracted from files that are highly important to the user, for instance, PDF files, e-mails, or word processor files. For other files like system help documents or library files, only a minimum of metadata is extracted. Clearly, those files that have metadata available can much easier be found than other resources. Secondly, *Beagle*$^{++}$ produces longer result lists, because more resources like people, publications or conferences are extracted that now can also be found. Comparing ambiguous to clear queries, the performance of *Beagle*$^{++}$ degrades a little bit. However, it is still better than *Beagle*.

Our enhanced ranking using ObjectRank further improves the quality of ambiguous and pure metadata queries. In these cases, contextual information seems to be of importance to identify highly relevant results. The context of search results seems to add noise for clear queries which degrades the quality. However, for large values of k, our *Beagle*$^{++}$ system with ObjectRank still outperforms its baseline system *Beagle*.

The poor performance regarding hybrid queries can be explained that the high expressiveness of these queries restricts the focus of results to such an extent that if no perfect results exist, fewer similar results are retrieved so that only smaller precision scores can be achieved.

Query type	P@1		P@2		P@3		P@4		P@5	
	B	B^{++}	B	B^{++}	B	B^{++}	B	B^{++}	B	B^{++}
Clear	0.85	**0.91**	0.73	**0.89**	0.68	**0.89**	0.66	**0.89**	0.67	**0.87**
Ambiguous	0.78	**0.82**	0.72	**0.84**	0.74	**0.83**	0.69	**0.84**	0.69	**0.83**

Table 3.2: Precision at top one to five for *Beagle* (B) and *Beagle*$^{++}$ (B^{++}).

Query type	P@1		P@2		P@3		P@4		P@5	
	B^{++}	B^{++}_{OR}	B^{++}	B^{++}_{OR}	B^{++}	B^{++}_{OR}	B^{++}	B^{++}_{OR}	B^{++}	B^{++}_{OR}
Clear	0.91	0.72	0.89	0.77	0.89	0.79	0.89	0.79	0.87	0.78
Ambiguous	0.82	**0.82**	0.84	**0.86**	0.83	**0.88**	0.84	**0.86**	0.83	**0.83**
Structured	0.92	**0.92**	0.85	**0.88**	0.83	**0.89**	0.83	**0.88**	0.83	**0.87**
Hybrid	0.68	**0.77**	0.73	**0.73**	0.74	0.70	0.72	0.65	0.67	0.63

Table 3.3: Precision at top one to five for *Beagle*$^{++}$ (B^{++}) and *Beagle*$^{++}$ with ObjectRank (B^{++}_{OR}).

Our performance evaluation results in Table 3.4 shows that our extension adds significant effort to the indexing and querying phase. However, indexing is performed only once; more precisely, when the user starts the Desktop Search engine the first time. Afterwards, files are indexed incrementally, whenever a file is modified, deleted, or created. Further, the indexing happens in the background with minimal priority on I/O operations. This minimally obstructs the user in using the Desktop computer. Therefore, the increase of indexing time is of low significance.

In contrast, the performance of the query phase is of high importance. After submitting the query to the system, the user waits for results. Here, the average response time of 2.2 seconds is still at a reasonable level. A significant proportion of the delay is caused by the coupling of *Beagle*$^{++}$ and the semantic store provided by NEPOMUK, and not by the search itself. A tighter integration of both parts would take advantage of existing performance improvement potential.

System	Indexing	Querying	avg. results
Beagle	39.21 m	0.348 s	5.714
Beagle^{++}	149.24 m	2.192 s	18.156

Table 3.4: Average indexing and querying time for *Beagle* and *Beagle*$^{++}$.

3.8 Lessons Learned and Public Perception

In the following we recapitulate the lessons we have learned, the open challenges that our prototype revealed, as well as the perception of our project in the Open Source community:

- By extracting complex metadata from Desktop data, we can improve Desktop Search in quality while adding a reasonable amount of effort to the indexing and querying phase. We can improve the user satisfaction for the task of Desktop Search compared to a baseline Desktop Search engine.
- We can further see that with a proper unique identification strategy and shared ontologies, a modular metadata extraction framework can provide fragments of complex metadata that nicely integrate into a central Semantic data store.
- We learn that with the addition of expressiveness — as hybrid queries provide it — the user has to cope with a more complex query articulation process. Here, the user has to be supported in order to construct valid and sensible queries. The articulation of a structured query using a text field is very limited and recognized as cryptic.
- With the construction of more complex hybrid queries, the evaluation of these queries expose performance problems of state-of-the-art semantic data stores. Firstly, further improvements of the performance into this direction are needed. Secondly, developers that aim at creating semantic applications need to identify the semantic data store that best fits the prospective work load induced by the user and applications. For this, means are required to make semantic stores comparable.
- In situations where the number of results for a query exceeds the cognitive capacity of the user, possibly important results do not come to the user's mind if they are too far down the result list and results recognized by the user are quite similar. Then, the user does not expect other results to come, loses interest and abandons the results list too early. Search result diversification is an effective means to avoid this risk of query abandonment, which should also be incorporated into semantic search.

We have been supported in our opinion that semantification of Desktop Search improves user satisfaction and attracts user interest at several occasions.

The *Gnome Beagle* project. We based our Semantic Desktop search infrastructure on the *Gnome Beagle* project. As we successfully proceeded with our implementation, the developers of this open source project approached us to integrate some of our achievements back into their project. We could extend *Beagle* to provide a semantic query end-point so that it can be queried as if it was an RDF store. For this, we developed a transparent translation between the Lucene documents and RDF resources serialized into RDF statements.

Tracker The Semantic Desktop search and storage project MetaTracker (also known as Tracker) is based on Semantic Web technologies as it uses RDF and NEPOMUK ontologies. Our *Beagle*$^{++}$ predates this open source community effort.

KDE A part of the NEPOMUK project was the active deployment of Semantic Web technologies into the KDE open source community[33]. Two years after the successful end of NEPOMUK, the KDE project continues to use NEPOMUK technologies and software. A semantic infrastructure is installed on every KDE Desktop today. Evidently, the open source community continues the semantification efforts stimulated by our activities in the NEPOMUK project and by developing the *Beagle*$^{++}$ prototype.

Further information on the *Beagle*$^{++}$ Semantic Desktop search infrastructure can be found at the *Beagle*$^{++}$ website[34]. Our website has been visited by 1,500 distinct IP addresses over the last year (as of December 1st, 2010), excluding web search engine crawlers. Our *Beagle*$^{++}$ system has been downloaded in that period of time almost 125 times. Thus, the interest in our system is still at a significant level, though the last version was released almost three years ago. A live-captured demo was uploaded to YouTube[35], which has been watched over six thousand times.

3.9 Conclusion and Outlook

I proposed a flexible architecture for a Semantic Desktop Search engine and presented its implementation in the *Beagle*$^{++}$ system. This architecture allowed several L3S members to implement and easily integrate their algorithms and semantic applications into this semantic infrastructure to successfully develop a prototype implementation of their research work. Further, a number of other research projects could be provided with Semantic Desktop data [47–49, 64, 177, 178].

Our Semantic Desktop Search implementation builds upon the open source *Beagle* system. I provided details about the new components: the Metadata Filters, the central RDF Repository, the Metadata Enrichment Components, its pool of enrichment applications (Entity Identification, ObjectRank and Attachment-File Linker), as well as the RDF Visualizer.

Finally, we have achieved a significant impact on the open source Desktop Search community regarding the introduction of Semantic Web Technologies. Besides the *Gnome Beagle* project that *Beagle*$^{++}$ bases on and the KDE Desktop, another Desktop Search project *Tracker* was initiated that builds on NEPOMUK ontologies and Semantic Technologies. Additionally, our demo video and the *Beagle*$^{++}$ website still experiences vivid attention.

Open Challenges The introduced Semantic Search engine is a semantic application that uses semantic technologies and imposes particular requirements on these technologies. From the implementation and usage of this prototype, I identified the following challenges that Semantic Search poses to state-of-the-art technologies used in the prototype. In the subsequent chapters of my thesis, I face these challenges by improving several fields of Semantic Search, in particular the following:

[33] KDE 4.0 Semantic Desktop Search and Tagging: `http://www.w3.org/2001/sw/sweo/public/UseCases/Nepomuk/`.
[34] *Beagle*$^{++}$: `http://beagle.l3s.de/`.
[35] YouTube – *Beagle*$^{++}$: `http://www.youtube.com/watch?v=Ui4GDkcR7-U`.

Structured query construction: The articulation of structured queries proved to be difficult for users in order to search their personal structured data (knowledge) in a structured way. Chapter 4 elaborates on a graphical structured query articulation approach that enables users to easily construct structured queries combined with keywords (hybrid queries).

Evaluation of complex hybrid queries: Once being articulated, the system has to evaluate hybrid queries against the stored structured data. In case of a particular class of hybrid queries, current stores have difficulties to efficiently evaluate them. In Chapter 5, I present improvements in the area of hybrid query evaluation.

Benchmarking hybrid queries: Though a number of semantic stores exists that support evaluation of hybrid queries, their performance with respect to particular types of structured queries usually varies. I present the first full-text search benchmark for semantic stores, which allows for extensive testing and comparison of semantic stores, as I will report in Chapter 6.

Diversification of Search Results: A last improvement in the area of Semantic Search in particular and search in general is presented in Chapter 7. The diversification of search results is known to improve user satisfaction by providing a short list of relevant *and* diverse results, rather than having users to scroll through long lists of only relevant but usually very similar results. Here, the diversification of a large number of results can be achieved in a streaming-based approach, the first one reported to this extent in the area of result diversification.

Chapter 4

Semantic Query Construction

During development, experiments and usage of our Semantic Desktop search engine *Beagle*$^{++}$, we have observed how challenging a textual interface for semantic queries is both for query interface developers and Desktop users. Consequently, we further studied means to allow users to articulate semantic keyword queries more easily. One approach proposes possible semantic meanings for a keyword query given by the user [127, 192]. Based on the ontology and occurrences of keywords, it constructs and ranks valid semantic queries. In contrast to this, we additionally developed a graphical interface that allows users to *explicitly* articulate structured queries for semantic search [107], which I describe in detail in this chapter.

We designed our graphical interface to achieve the following objectives. While constructing queries, the user *learns* about the structure of the Desktop data. This is important and necessary, since we decided for implicit metadata extraction (see Section 3.4 on page 38) where predefined ontologies are used and the user initially does not know the structure of the Desktop data. Even in case the user explicitly articulates knowledge — she creates her personal data model and therefore knows it best — our interface *reminds* the user of the actual structure. The main objective, however, is to keep the query articulation free from any kind of syntax, which otherwise would allow for a multitude of possible syntax errors in the query articulation process.

Finally, the user should not be able to construct structured queries that do not make sense (from the ontology's point of view). For instance, an e-mail will never have a birthplace, whereas a person does. And a person does not have a subject, but e-mails do. Such queries would not provide any results, as they do not comply with the ontology and data. Therefore, evaluating them is of low value for the user. In particular, this chapter makes the following contributions:

1. I propose a formal incremental structured query construction model. This model is implementation-independent and constitutes the basis for our graphical interface.

2. Further, I present a joint work graphical user interface that implements our query construction model. The interface implementation is integrated into the NEPOMUK Social Semantic Desktop [131] [166, Sec. 2.10.2], which is available for free download[1].

3. The architecture of this graphical interface allows to plug in different recommender algorithms that provide the user with useful query construction steps to reduce user effort.

The remainder of this chapter is organized as follows. After a literature review of existing visual query systems, I present a formal definition of our query construction model. The second part of this work comprises the implementation of our graphical user interface that uses this model. I complete this chapter with conclusions and outlook.

[1] NEPOMUK Eclipse: `http://nepomuk-eclipse.semanticdesktop.org/`.

4.1 Related Work

A number of user interfaces for semantic query construction have been proposed in recent years. These can generally be categorized into graphical [32, 41, 90, 155], keyword-based [3, 113, 180, 185, 196], and natural language interfaces [27, 106, 115]. While the latter are deemed to be most convenient for users [106], these either can misinterpret queries or still require the use of proper terminology and are thus rather controlled language interfaces. Keyword-based interfaces for the Semantic Web [113, 185, 196] and databases [3, 180] construct possible interpretations and find the most probable one via ranking. While this interpretation takes cognitive load off the user, it also takes away a significant proportion of control. We, in contrast, aim at allowing people to explicitly articulate the semantics of their information need.

Several graphical query systems exist. The first one was Query-By-Example (QBE) [198], developed in the late 1970s. Below, we discuss some significant works in this area, mainly diagram-based systems [40] that use the same approach as we do. A remarkable early visualization approach of RDF queries was done by Harth *et al.* [90]. The authors sketch the piece-by-piece construction of a SPARQL query and the possible visualization of these pieces. However, this is rather a graphical notation than a query construction system. Due to the piece-wise graphical translation, this notation contains the same technical complexity and terminology as SPARQL, which we want to hide from the user in our system.

iSPARQL[2] is a powerful tool to specify all kinds of SPARQL queries. However, to formulate even a simple query, the user must know technical concepts such as variable and data type properties. NITELIGHT [156] is similar to iSPARQL. The main difference between them lies in the visual notations adopted by each of them. Regardless of all the expressiveness of these tools, their visual notations are not suitable for the profile of Semantic Desktop end-users.

SEWASIE [41] uses an interface closer to the mental model of users allowing them to make a limited set of queries. Further, during the query formulation, the user is guided by a recommendation system based on the ontology of the data domain. Conceptually, this work is very similar to what we do but little conclusive due to limited information contained in the paper.

Yet SPARQLViz [32] uses a completely different approach from ours, in which the user draws up a query through a sequence of forms. Despite the lack of empirical studies between the two approaches (diagram-based vs. form-based query), the model adopted by SPARQLViz requires the user to have an advanced technical knowledge because it is more focused on making complex queries rather than providing a simple interface to do the same.

It has been identified in previous works [17, 20, 181] that a progressive modeling search activity can provide better results than a single, static search. Yet, strong methods and interaction mechanisms are still missing for searching the Semantic Desktop. In this work, we fill this gap by presenting a methodologically designed interface for Semantic Desktop Search that combines and exploits current interaction mechanisms. Our approach takes into account what the user knows (the vocabulary and the mental model), while it respects what the user does not know (the data structure and query languages), to finally give her what she requests.

[2]OpenLink iSPARQL: `http://demo.openlinksw.com/isparql/`.

4.2 The Query Construction Model

The core of our graphical query interface is our query construction model. It defines what constitutes a query and which constructive operations can be performed on it. We design an iterative construction process in which each possible transition from the current query to a more complex query can be achieved by a small set of operations.

The Query Construction Model Our query construction model $M = (\mathbf{\Omega}, \mathbf{Q}, \mathbf{O})$ consists of the ontology $\mathbf{\Omega}$, the space of all possible queries $\mathbf{Q}$ that comply with the ontology, and operations $\mathbf{O}$ that can be performed on the queries $q \in \mathbf{Q}$.

The Ontology An ontology $\mathbf{\Omega} = (\mathbf{T}, \mathbf{P}, \mathbf{s_T}, \mathbf{s_P})$ contains types (classes) $t \in \mathbf{T}$, properties (relations) $p \in \mathbf{P}$, as well as sub-type $\mathbf{s_T}$ and sub-property $\mathbf{s_P}$ functions that relate types and properties as generalization or specialization to each other. Properties connect a pair of types and are defined as $\mathbf{P} \subseteq \mathbf{T} \times \mathbf{T}$ where the property connects the first type to the second type. The sub-type and sub-property functions are defined as $\mathbf{s_T} : \mathbf{T} \to \mathbf{T}$ and $\mathbf{s_P} : \mathbf{P} \to \mathbf{P}$. These functions provide a mapping from types to types $\mathbf{s_T} \subseteq \mathbf{T} \times \mathbf{T}$ and from properties to properties $\mathbf{s_P} \subseteq \mathbf{P} \times \mathbf{P}$. They are represented as tuples $(t_1, t_2) \in \mathbf{s_T}$ and $(p_1, p_2) \in \mathbf{s_P}$. These mappings model that t_1 is a sub-type of t_2 and p_1 is a sub-property of p_2, respectively. They can be visualized as a directed edge from a vertex t_1 to vertex t_2, where all these edges form an acyclic, directed graph. These functions are considered to contain the full transitive closure (see Section 2.1), *i. e.*, $(x, y) \in \mathbf{s_T} \wedge (y, z) \in \mathbf{s_T} \Rightarrow (x, z) \in \mathbf{s_T}$ (the same holds for $\mathbf{s_P}$). In summary, an ontology $\mathbf{\Omega}$ defines types and properties that connect these, while types and properties can relate to each other as specification or generalization.

The Query A query $q \in \mathbf{Q}$ with $q = (\mathbf{N}, \mathbf{K}, \mathbf{R}, t_\mathbf{N}, t_\mathbf{R})$ consists of a number of nodes $\mathbf{N}$, keywords $\mathbf{K}$, relations $\mathbf{R} \subseteq \mathbf{N} \times (\mathbf{N} \cup \mathbf{K})$, as well as functions $t_\mathbf{N} : \mathbf{N} \to \mathbf{T}$ and $t_\mathbf{R} : \mathbf{R} \to \mathbf{P}$. These functions provide links from nodes ($t_\mathbf{N}$) and relations ($t_\mathbf{R}$) to the ontology. Since a query has to comply with the ontology, the possible types of nodes that are part of a relation are restricted by the ontology, namely, the types that constitute that property. Formally, for nodes n_1 and n_2 of relation $r = (n_1, n_2)$ the following must hold:

$$\begin{array}{rcl} (r, p_r) \in \mathbf{t_R} & \wedge & p_r = (t_{n_1}, t_{n_2}) \quad \wedge \\ (n_1, t_{n_1}) \in \mathbf{t_N} & \wedge & (n_2, t_{n_2}) \in \mathbf{t_N} \end{array} \tag{4.1}$$

Under consideration of the sub-type and sub-property relations, this extends to

$$\begin{array}{rcl} (r, p_r) \in \mathbf{t_R} & \wedge & (p_r, (t_{p,1}, t_{p,2})) \in \mathbf{s_P} \quad \wedge \\ (n_1, t_{n_1}) \in \mathbf{t_N} & \wedge & (n_2, t_{n_2}) \in \mathbf{t_N} \qquad \wedge \\ (t_{n_1}, t_{p,1}) \in \mathbf{s_T} & \wedge & (t_{n_2}, t_{p,2}) \in \mathbf{s_T} \end{array} \tag{4.2}$$

In our query model, nodes and relations can also be un-typed. In this case, the nodes or relations always satisfy Equation 4.2.

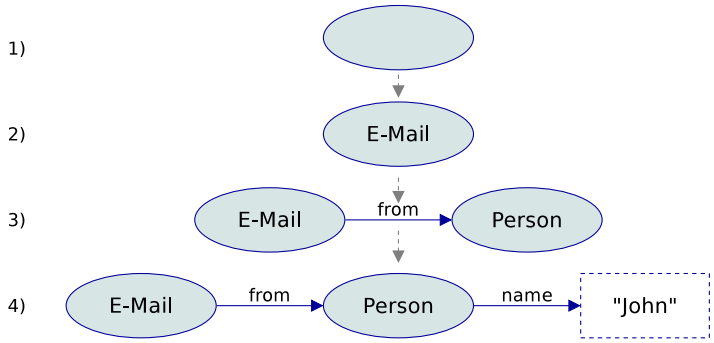


Figure 4.1: The incremental creation of the query *all **E-Mails** from a **Person** called „John“*: 1) a single un-typed node, 2) the node is typed as an e-mail, 3) the e-mail is connected to a person via `from`, 4) the person is connected via `name` to a keyword.

The Operations Each operation $o \in \mathbf{O}$ produces a new query out of the argument query, that is again element of the query space: $o : \mathbf{Q} \rightarrow \mathbf{Q}$. In the following, we define the operations of our incremental query construction model, where the initial query is $q_0 = (\{n_0\}, \emptyset, \emptyset, \emptyset, \emptyset)$.

Typing a node: $o_{tn} : \mathbf{Q} \overset{n,t_n}{\rightarrow} \mathbf{Q}$ The type of a node can be set by this operation, or, if already set, can be changed. This operation is only valid for types that produce queries that still comply with the ontology (see Equation 4.2).

Input: $q = (\mathbf{N}, \mathbf{R}, \mathbf{K}, \mathbf{t_N}, \mathbf{t_R}) : n \in \mathbf{N}, t_n \in \mathbf{T}$.
Result: $q' = (\mathbf{N}, \mathbf{R}, \mathbf{K}, \mathbf{t_N} \setminus \{(n, \cdot)\} \cup \{(n, t_n)\}, \mathbf{t_R})$.

Typing a relation: $o_{tr} : \mathbf{Q} \overset{r,p_r}{\rightarrow} \mathbf{Q}$ The type of a relation can be set or changed by this operation. However, it is only valid for relation types that produce queries that still comply with the ontology (see Equation 4.2).

Input: $q = (\mathbf{N}, \mathbf{R}, \mathbf{K}, \mathbf{t_N}, \mathbf{t_R}) : r \in \mathbf{R}, p_r \in \mathbf{P}$.
Result: $q' = (\mathbf{N}, \mathbf{R}, \mathbf{K}, \mathbf{t_N}, \mathbf{t_R} \setminus \{(r, \cdot)\} \cup \{(r, p_r)\})$.

Adding a relation to another node: $o_{an} : \mathbf{Q} \overset{n_1,n_2,p}{\rightarrow} \mathbf{Q}$ This operation adds a new relation $r = (n_1, n_2)$ of type p to the query. It connects the two nodes n_1, n_2, where at least one of them already has to be in $\mathbf{N}$: $n_1 \in \mathbf{N} \vee n_2 \in \mathbf{N}$. It is easy to show that if exactly one node has to be in $\mathbf{N}$, then this would restrict $\mathbf{Q}$ to contain acyclic queries only. Again, p and the types of n_1, n_2 have to comply with the ontology.

Input: $q = (\mathbf{N}, \mathbf{R}, \mathbf{K}, \mathbf{t_N}, \mathbf{t_R}) : n_1 \in \mathbf{N} \vee n_2 \in \mathbf{N}, p = (t_{n_1}, t_{n_2}) \in \mathbf{P}$.
Result: $q' = (\mathbf{N} \cup \{n_1, n_2\}, \mathbf{R} \cup \{r\}, \mathbf{K}, \mathbf{t_N} \cup \{(n_1, t_{n_1}), (n_2, t_{n_2})\}, \mathbf{t_R} \cup \{(r, p)\})$.

Adding a relation to a keyword: $o_{ak} : \mathbf{Q} \overset{n,k,p}{\rightarrow} \mathbf{Q}$ This operation adds a new relation $r = (n, k)$ of type p to the query that connects node n with keyword k, with $n \in \mathbf{N}$.

Input: $q = (\mathbf{N}, \mathbf{R}, \mathbf{K}, \mathbf{t_N}, \mathbf{t_R}) : n \in \mathbf{N}, p \in \mathbf{P}, k$.
Result: $q' = (\mathbf{N}, \mathbf{R} \cup \{r\}, \mathbf{K} \cup \{k\}, \mathbf{t_N}, \mathbf{t_R} \cup \{(r, p)\})$.

Arguably, a possible fifth operation is the removal of a relation. While the other four operations construct queries that comprise one single connected graph, such a removal operator could create a query with multiple connected components. Though such a query is valid, its usefulness for users is low. Its result set can potentially explode in size since it is the permutation over the result sets of the connected components. Such kind of a query can also be constructed by o_{an} if none of the nodes need to be in $\mathbf{N}$ already. We explicitly exclude such kind of queries.

We designed deletion as a temporal phenomenon where undoing the addition of relations sufficiently allows for deletion. This is therefore a usability feature and consequently provided by the graphical user interface (GUI) implementation.

These four operations allow for the creation of any directed labeled graph-like query, *i. e.*, any possible conjunctive query, and is thus not restricted to acyclic graphs. However, our GUI implementation will, for the sake of simpler visualization, only allow for the creation of acyclic queries. See Figure 4.1 for an example query transition following the above described model.

4.3 User Interface

Our graphical user interface (GUI) provides the user with a visualization of the current query, means to perform operations that extend this query, as well as an undo function. Recommendations provide possible semantic terms to extend the query that comply with the ontology. Keyword filters allow to quickly find semantic terms in case the list of options is rather long.

We restricted the addition operation in our implementation to only connect an existing node with a new one. Therefore, loops cannot be created, only tree-like queries can be constructed. This simplification of the user interface should be sufficient in most cases. However, in cases where such cycles are required, it is only a matter of extending the interface since the query construction model natively supports loops.

The interface contains three main canvases: a construction area showing the current query, a node/relation type list, and a relations list. The graphical representation of nodes and relations is straightforward and similar to the modeling language. We use a box representation for the node containing its type, whereas incoming and outgoing relations are represented by oriented arrows. Note that un-typed nodes and relations are represented without a label. Keywords are shown as a text field in which the user can enter the actual keyword query.

The tree-like query is represented horizontally. The initial node is the left-most root of the tree. All paths are drawn in a depth-first manner from left to right, branching in chronological order of their creation from top to bottom. One node or relation is always selected. Two lists provide options for semantic extension of node types or relations, depending on the actual selection. For node and relation selections, all allowed types are listed. For nodes only, all relations allowed for addition are given. When the user selects a type, the selected node or relation is typed accordingly. When a relation is clicked on, it is added to the selected node and a new node is created.

The graphical user interface is implemented using HTML, JavaScript and Cascading Style Sheets (CSS). These lightweight implementation technologies bring the benefit of modularity and allow for easy design tailoring.

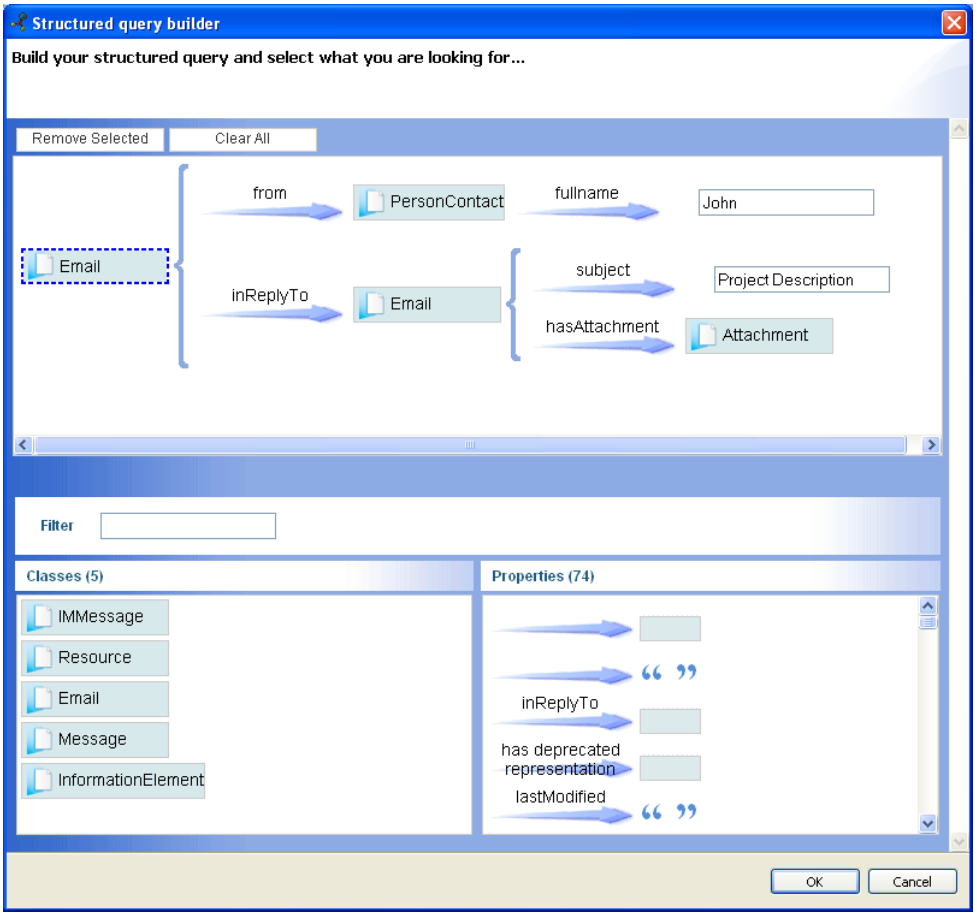


Figure 4.2: The graphical user interface (GUI) for our query construction model.

Implementation

In the following, I focus on the architecture of our implementation and interesting details. We decided for a two layer architecture which separates the *presentation layer* from the *query construction model layer*. The latter layer wraps the data and its ontology, and implements the iterative query construction. The presentation layer uses operations of the query construction model layer to provide query construction actions to the user. It collects and forwards feedback to the query construction layer in order to proceed to the next construction step.

This separation allows for easy plug in of different presentation layer implementations (*i. e.*, Graphical User Interfaces) on top of the query construction layer. Further, the GUI does not require any knowledge about the data itself, it only knows the graphical query construction process. The construction model in return does not care about presentation of operations, recommendations and graphical queries.

Presentation Layer — The GUI For the implementation of the proposed interface we followed the approach to add semantic annotations to the HTML[3] code to define the behavior of the interface widgets. Each behavior is dynamically added depending on the current state of the query construction process. We used the Prototype[4] library which allows to easily navigate the HTML Document Object Model (DOM)[5] tree, select elements by their class attribute values and link operations to interface events like `onClick` or `onMouseOver`. This technique enables us to create a very dynamic query construction interface with continuous representation, incremental actions and feedback.

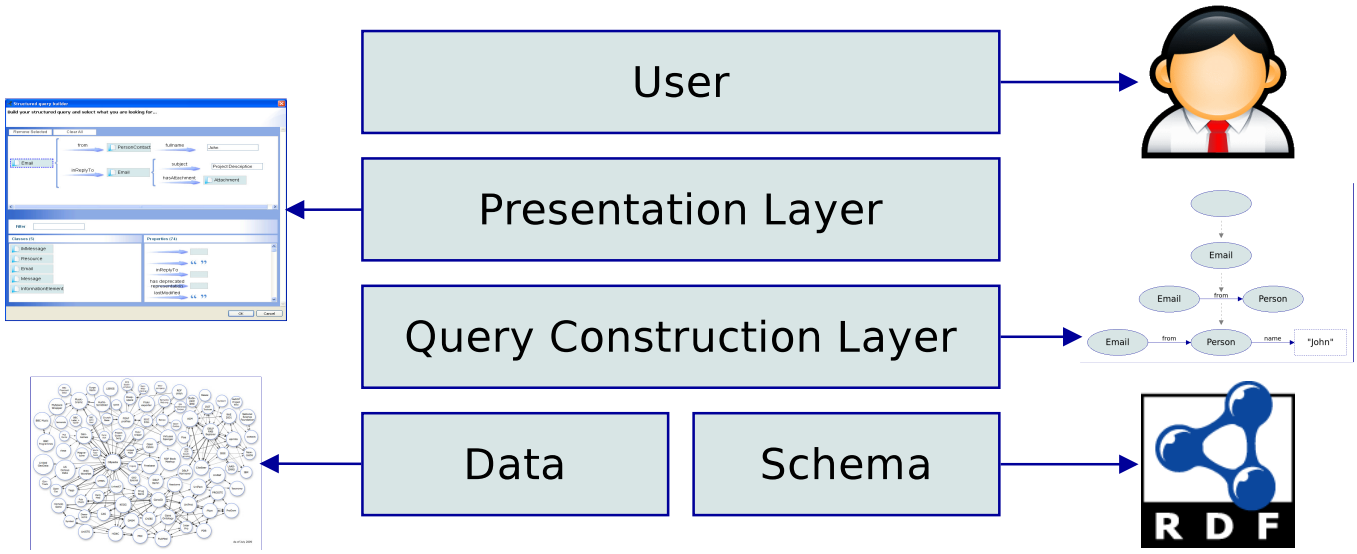


Figure 4.3: The layered architecture of our incremental query construction user interface.

Query Construction Model Layer The query construction model layer provides two services to the presentation layer. Firstly, it recommends types and relations, secondly, it modifies the current structured query based on the user's action, as described earlier.

After each of these actions, the current structured query object is updated and provided to the presentation layer, which in return updates the visualization of the query, retrieves new recommendations and updates the option lists. This cleanly separates visualization and user actions from the query construction model and the underlying ontology. The final query represents a syntactically correct and ontologically valid SPARQL query which can then be evaluated against the stored data.

[3]W3C XHTML2: `http://www.w3.org/MarkUp/`.
[4]Prototype JavaScript Framework: `http://www.prototypejs.org/`.
[5]W3C Document Object Model: `http://www.w3.org/DOM/`.

Type and Relation Recommendations

The recommendation of valid types and relations is crucial for the interface to be beneficial to the user. To control the amount of information and options presented to the user in every construction step, good recommendations are essential. If the underlying ontology is sufficiently large, recommendations can contain hundreds of (sub- or super-) types or relations. Scrolling through such a long list is obviously not desired by the user.

The proposed keyword filter is a first helpful feature to handle such large collections of options. However, reducing the number of options per step to a handy list, even if then the number of steps for the complete query construction increases, is a desirable goal. The complete list of options for a type (or relation recommendation) contains all super- and sub-types (super- and sub-relations). This occasionally large list can be reduced to:

1. direct super- / sub-types
2. only types that actually occur in the data
3. closest super- / sub-types that have instances
4. only meaningful types

The sub-types and sub-relations hierarchies can be exploited to reduce the number of choices per step. While (1) decomposes any step from one type to any of its super- or sub-types into as many steps as there are intermediate types, (2) and (3) reduce this number to useful target types. However, if all types have instances, this degrades to showing all super- and sub-types.

Type hierarchies sometimes contain "engineering" types that serve technical purposes, rather than representing any meaningful concept. For instance, the PIMO ontology used for our prototype (see Section 3.4.3) has a `pimo:Thing` type as the root of any meaningful type. It has the super-type `pimo:ClassOrThing` to allow for adding the same relation to PIMO classes and PIMO things. This technical type is clearly meaningless for the user. In generally, one can say that the closer a type is to the root of the type hierarchy, the more likely it is to be of such a general nature. Those types should be hidden from the user (4). However, it is not a trivial task to detect at which level of the hierarchy the meaning degrades. For a given ontology, this selection can be done by an ontology expert.

On top of this filtering, the order can push more relevant types to the top of the list and thus push them faster into the awareness of the user. Such an ordering could be based on:

1. the number of existing instances per type
2. a utility estimation metric
3. the usage frequency during previous queries

The described recommendation strategies provide further room for improvement as do not we make use of any ranking or elaborate on its potential to improve the user's performance of solving the task of query construction.

4.4 Conclusion and Outlook

I presented our simple and powerful graphical user interface that allows to visually construct structured SELECT SPARQL queries through direct manipulation. In this work, functionality and simplicity is balanced in a way to provide the common user means to interact with the ontology without the usual cognition load of a semantic querying tool.

The clean architecture allows for easy employment of additional sophisticated semantic recommendations to reduce user effort and increase user performance of the task of structured query construction. A usability test for evaluating the GUI's efficiency is considered future work.

Chapter 5

Evaluating Hybrid Queries

More and more applications use the RDF framework as their data model and RDF stores to index and retrieve their data [94]. With the help of appropriate graphical user interfaces (see Chapter 4 on page 59), structural queries combined with full-text search, so called *hybrid queries* [28], can be constructed by users. We agree that hybrid queries are unlikely to be articulated by lay users [95, p. 38], but the same user might want to invest more effort in another information need that is of more value and thus justifies higher costs. However, existing mature RDF stores expose performance issues handling *complex* hybrid queries. With complex, I reference a class of hybrid queries that contains *multiple* full-text search operations referring to *distinct* resources. Further, such RDF stores provide only a limited set of full-text search features.

In the following, I report on my work that extends an existing RDF store with full-featured full-text search in a generic way that can be adopted to other RDF systems [124]. Additionally, I apply store-specific improvements that allow for more efficient evaluation of hybrid queries to better support semantic applications [192]. My work comprises the following contributions:

1. I present means of adding full-text search semantics to SPARQL queries without modifying the SPARQL syntax. This ensures that the approach uses queries that are compliant with the W3C recommended SPARQL query language [146].

2. I report on a unified framework combining structured and full-text indexing as well as searching RDF data, utilizing Semantic Web and Information Retrieval technologies.

3. I outline in-depth store-specific improvements achieved during implementation and usage of the full-text search enabled RDF store used in multiple research projects [122,131,165].

In particular, these improvements contain:

An accurate cardinality estimation for statement patterns which are evaluated as range queries over on-disk B-trees [75, Sec. 13.3.4]. The number of results such a range query spans is estimated based on two look-ups and general statistics only.

A better join order strategy is implemented that uses these cardinality estimations and efficiently approximates the optimal query plan better than the existing query planner.

The optimization of on-disk B-trees to improve performance of the most frequent B-tree operation: range queries. This operation comprises look-ups and iterations which I optimize by minimizing the number of input-output (I/O) operations and disk seek distances.

A better index selection strategy addresses the dynamic access to B-tree indices imposed by subsequent range queries of the query plan.

The effectiveness of these improvements is confirmed with three performance evaluations each having a specific scope: the pure full-text search performance of the whole framework, the improved on-disk index structure performance that relates to the semantic part of a hybrid query, as well as the general performance of hybrid query evaluation. Additionally, a more comprehensive comparative evaluation is performed in Chapter 6 on page 103. There, a synthetic benchmark specifically designed to study the performance of hybrid queries is presented and evaluated against three additional well-known full-text search capable RDF stores. Results quantify how challenging complex hybrid queries are to those competing systems.

The implementation called *LuceneSail*[1] is available as free open source software and can be used for many existing Sesame RDF storages, *e. g.*, File-, MySQL- and PostgreSQL-based ones. The principal concept is general enough that it can even be adopted to other existing RDF storages. LuceneSail was used in the research projects *Beagle*$^{++}$ [122], Gnowsis [165], NEPOMUK [131], and several research activities [46, 64, 168]. It is further used in the commercial software Aduna Autofocus and the news domain: ABC News[2] uses LuceneSail to power many features across the site, which as of August 2009 rose with over 200 million page views per month to the 5th rank in Nielsen rankings of online news sites[3].

5.1 Related Work

Structured query languages such as SQL or SPARQL [146] are known to provide high expressivity regarding the *structure* of data, whereas full-text queries target at the *content*. Integrating full-text search (IR) [12, 117] with structured search (DB) [75] is an actively discussed undertaking [5, 10, 43]. A lot of research has been conducted in DB fields such as relational databases [63, 97], XML databases [57] and RDF stores [28, 195], to name only a few.

Full-text search clearly adds a large set of rich features to SPARQL, *e. g.*, boolean, phrase, wildcard, and proximity queries [12, Ch. 4] (see Section 5.2.1). A relevance score and so called *snippets* provide that the textual context of the matching keywords can be retrieved.

While SPARQL [146] offers a high expressiveness with respect to structured queries, full-text related queries are not well supported. There are only two features that target on textual content of RDF graphs: regular expression and full string matching. Interestingly, none of the widely used RDF stores support the standardized feature efficiently, *i. e.*, with any kind of fast index structures [11, 51]. This would allow for an early evaluation of regular expressions that potentially reduce intermediate results. In contrast, this standardized feature is rather implemented as a filter at a late stage of query evaluation, which is much easier. I believe the reason is that regular expressions are more expressive than commonly required, thus an exhaustive efficient implementation does not pay off. It is rather exploited for specific purposes only. In [143], for instance, regular expressions are used to evaluate arbitrary path expressions between resources. They are evaluated in an efficient way which, however, is limited to this particular application.

[1] The Sesame2 LuceneSail: `http://dev.nepomuk.semanticdesktop.org/wiki/LuceneSail`.
[2] ABC News: `http://abcnews.go.com/`.
[3] According to e-mail conversation with an ABC News software developer on Aug 17, 2009.

In contrast, many popular RDF stores support efficient keyword search, putting significant effort in a feature that is *not* part of the W3C Recommendation. Some RDF indexing systems even consider the combination of inverted keyword indices and statement indices as fundamental building blocks for efficient RDF indexing [91]. This investment signifies the high demand of Semantic Web applications for RDF stores to support full-text search or hybrid queries.

A large spectrum of RDF stores is available today. The most promising and most developed ones have already been reviewed in several surveys. A first and general survey can be found in [184], a more relational database-focused survey is [19]. A performance analysis of some file, MySQL- and in-memory-based RDF stores was done in [112].

The first store employing an inverted index on literals is the RDFStore [149]. Full-text search is incorporated into RDQL and their RDFStore programming interface. One inverted index is maintained for all literals, together with basic affix stemming. Predicate specific full-text search is achieved by joins, which the authors expected to be cheap since they use compressed sparse bitmaps. There is no ranking provided and only basic keyword queries can be issued.

Yet Another RDF Store (YARS) [89, 91] also uses an inverted index for all terms and provides simple full-text queries but neither stemming nor scoring. Since there is only one index for all literals, full-text queries specifying a predicate require one additional join operation. Like our approach, YARS uses virtual properties, but without any feedback from the full-text search. Furthermore, YARS uses queries in N3 notation [171, p. 72], which is not commonly done.

The University of Southampton developed 3store [88], a MySQL-based RDF store implemented in C. MySQL provides a full-text index for each column, allows simple keyword queries, boolean operators, phrase queries and some kind of query expansion[4], as well as a relevance measure. Thus, the 3store gains its full-text capabilities from MySQL, which is therefore not general enough to be applied to other RDF stores.

An RDF storage that incorporates Lucene quite similarly to our work is Kowari [188]. When the Lucene support is used, all literal values in queries are interpreted as Lucene queries and a matching score can then be received. In contrast, the semantics of textual data are clearly defined in LuceneSail due to *virtual properties* (see Section 5.3). Kowari introduces the Interactive Tucana Query Language (iTQL) and implements a subset of RDQL only.

With the help of inverted indices for RDF literals, several effective solutions are available [89, 91, 149]. However, these still lack lots of features provided by state-of-the-art IR systems. Existing RDF stores with full-text search capabilities either employ non-standard query languages [89, 91, 149] or the semantic of a query depends on the configuration of the RDF store [37, 188]. Moreover, they do not provide the query expressiveness of state-of-the-art information retrieval (see Section 5.2.1) and relevance or snippet information for search results [37, 88, 188]. Most systems employ a full-fledged IR system, but do not exhaustively exploit or expose its capabilities. Our full-text extension demonstrates a combination of an RDF store and an IR system. It facilitates pure IR queries (full-text) within pure SPARQL queries (structured queries), taking full advantage of the expressiveness of each of them. This is achieved without any modifications of existing RDF query language syntax.

[4] MySQL 5.0 Reference Manual - Full-Text Searches with Query Expansion: `http://dev.mysql.com/doc/refman/5.0/en/fulltext-query-expansion.html`.

5.2 Structured and Full-text Indexing and Search

Today's RDF query languages and RDF stores lack sophisticated full-text search. This is surprising since literals give meaning to the Semantic Web: they are connecting humans with the data they use. The structure that is interconnecting the nodes is useless without the meaning of the nodes. Searching the Semantic Web greatly benefits from good literal search: it finds the most relevant resources regarding a desired meaning. Literal search filters out relevant nodes of all nodes matching specific structure constraints. In return, structure gives more meaning to nodes when relevant literals are found.

This work combines structured with full-text search, two fields of research that more and more coalesce into one topic of research [5]. In the following, I elaborate on these two fields and introduce an implementation library for each of them: *Sesame* [34] and *Lucene*[5].

5.2.1 Full-text Indexing and Search

In Information Retrieval, the Vector Space Model [12, Sec. 2.5.3] is the most popular retrieval model. Here, the plain text content of a document $\mathbf{d}$ is transformed into a vector of term weights $\mathbf{v}_d = (w_{1,d}, \ldots, w_{N,d})$. Each dimension $w_{i,d}$ refers to a unique term occurring in the document corpus. The notion of terms and term weights can be defined in various ways. Usually, simple words are used as terms, but named entities, phrases, and sentences are also possible for particular applications. Different forms of the same word can be mapped to the same term by stemming techniques (remove affixes to extract the word stem [12, Sec. 7.2.3]) or lemmatization (provides the word root called the *lemma*). This generally improves search quantity and quality as documents are found that do not contain the exact query word but a different word form.

As an example for the term weights $w_{i,d}$, the *TF-iDF* scheme (Equation 5.1) is generally accepted as state-of-the-art [12, Sec. 2.5.3] [117]. Here, the term frequency $tf_{t,d}$ of term t in document d is multiplied with the inverse (logarithm of the) term Document Frequency idf_t, which rewards infrequent terms and penalizes common terms *w. r. t.* the document corpus.

$$w_{t,d} = tf_{t,d} \cdot idf_t \quad , \qquad idf_t = \log \frac{|D|}{|\{\mathbf{d}' \in \mathbf{D} : t \in \mathbf{d}\}|} \tag{5.1}$$

Even though this Vector Space Model treats a document as a bag-of-word and ignores the position and order of terms, it yields to high quality document-document (similarity) and document-query (relevance) measures. The most commonly used one is the *cosine similarity*, which measures the similarity of two documents as the cosine of the angle between the respective N-dimensional term weight vectors (Equation 5.2). In case one of the document vectors is a term weight vector of query terms, it becomes a relevance measure.

$$cosine\ similarity(\mathbf{d_1}, \mathbf{d_2}) = \frac{\mathbf{d_1} \cdot \mathbf{d_2}}{|\mathbf{d_1}| \cdot |\mathbf{d_2}|} = \frac{\sum_{i=1}^{N} w_{i,d_1} \cdot w_{i,d_2}}{\sqrt{\sum_{i=1}^{N} (w_{i,d_1})^2} \cdot \sqrt{\sum_{i=1}^{N} (w_{i,d_2})^2}} = \tag{5.2}$$

[5] Apache Lucene: `http://lucene.apache.org/`.

The cosine is defined for the interval $[0, 1]$ where 0 identifies orthogonal vectors (they have no term in common) and 1 indicates parallel vectors (identical term frequency vectors). In between, this measure provides continuous degrees of similarity (or relevance), whereas in contrast, DB queries only provide binary relevance that indicates whether or not results satisfy the query.

To retrieve documents for a given query, an *inverted index* lists all documents that contain a single term in descending order of term weight. The top-k most relevant documents can efficiently be retrieved by merging the query terms' posting lists [12, Ch. 8].

Besides keyword queries, today's Information Retrieval systems provide a number of different kinds of queries that can be efficiently evaluated with an inverted index [12, Ch. 4]:

phrase queries where a term consists of more than one word

wildcard * and ? allow arbitrary sub-strings and characters

fuzzy queries finds terms that are similar to the given one

proximity queries terms may appear in a specified distance

range queries all terms that are alphabetically between two given terms

fielded queries a term has to occur in a particular part of the document

All those kinds of queries can be combined to more complex queries with the help of boolean operators (AND, OR, NOT). Further, terms of a query can be given different weights to reflect individual importance. Textual queries can contain mandatory, optional and prohibited terms. Further, with stemming/lemmatization, phrase, wildcard, fuzzy, proximity and range queries, Information Retrieval textual queries reflect their superiority in contrast to simple exact string matching as employed in today's graph query languages.

Fielded queries base on a document representation with multiple term frequency vectors, each for a different part of the document. The title of a document, for instance, may be separated from the rest of the document text. Then, one query term can be specified to be part of the title, whereas the rest of the query may match the body of the document.

Such fields allow to introduce simple semantics into IR search. Particular query terms can be given a specific meaning (title term, author name). Those semantics are comparable to properties pointing to literals in RDF. However, it is not possible to semantically interconnect documents. Since all documents are individually and independently modeled and stored in the IR index, IR queries can only express characteristics of one document and not of multiple ones. To exemplify this, the query

```
Books with "games" in their title,
written by an author who
also wrote books about "computer programming"
```

cannot be evaluated by an IR system that employs such a document model. Further, the author[6] itself cannot be retrieved as the model is centered around documents (books) and not associated entities (authors).

[6] A right answer is *Donald E. Knuth*: `http://www-cs-faculty.stanford.edu/~uno/books.html`.

Lucene: a Full-text Search Engine Apache Lucene[7] is an open source, pure-Java, high-performance, full-featured text search engine library with very low resource requirements. It implements all common IR features described above.

Lucene's core data structure is a Lucene document, which basically consists of a number of name-value fields. The documents are indexed by applying different strategies on each single field, such as tokenization, indexing, or simply storing. Depending on the strategy, Lucene employs inverted indices on the fields. Each retrieved document provides a score value reflecting its relevance based on *TF-iDF* term vectors and cosine similarity. If fields are stored, Lucene can provide previews — so called snippets — of the matching section. On query-time, the field to be queried must be specified, so it is not possible to simply query all fields at once. To work around this problem it is considered best-practice to index all fields' values in an additional field. This enables queries over all fields at the cost of increased index space.

5.2.2 Structured Indexing and Search

Search on structured data allows to formulate very specific and complex queries. The corresponding results are always a set of exact matches, rather than a ranked list as in IR. Since RDF resources typically provide a lot of textual information, IR search functionality clearly provides a lot of benefits to users. Studies have shown that users most probably start searching by file location or classification in an ontology and then use simple keyword searching [15], which gains importance if the Semantic Web shall get widely used. However, current graph query languages strongly focus on structural queries and neglect the expressiveness of textual queries. To support users both in file location (structure) and full-text (textual) search, a combination is needed.

Usual RDF query languages such as SPARQL [146] and RDQL [170] (see Section 2.1 on page 24) query a graph by means of graph patterns with wildcards and variables. The result set is either a set of graph fragments that exactly match the patterns, or a set of variable bindings (values). As usual in the structured data domain, results are always exactly matching the query.

Sesame: an RDF store Sesame [34] is an open-source framework for storage, inference and querying of RDF data, developed by the software company Aduna. Sesame is very modular and can be configured to use storage backends such as main memory, a relational database, or a native RDF storage file. This flexibility is achieved by stacking Storage And Inference Layers (SAIL). Each layer provides a particular service, *e. g.*, persistent on-disk storage, in-memory storage, or transparent RDFS inferencing. The combination of SAILs constitutes the actual Sesame store.

The addition and deletion of RDF data (RDF transactions) as well as querying is based on a JDBC-like[8] model where all operations are done through a connection. This connection-centered approach ensures clean handling of transactions and concurrent access to an RDF store.

In the following, I describe our approach of integrating Lucene queries into graph query languages in general, and specifically into Sesame, enabling both, complex textual queries and structural queries on RDF graphs.

[7] Lucene: http://lucene.apache.org/

[8] The Java abstraction layer for relational database connections, *i. e.*, Java DataBase Connections.

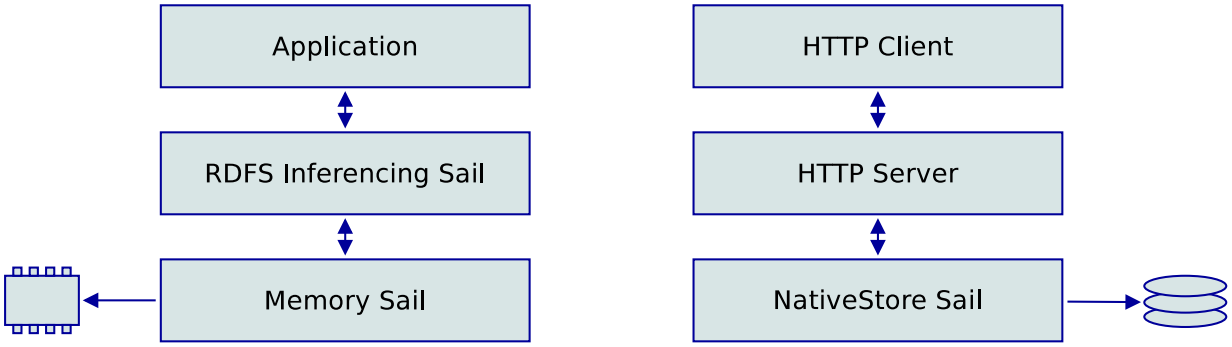


Figure 5.1: Sesame RDF store examples: left) memory-based store with RDFS inferencing embedded in an application, right) file-based native store, accessible via HTTP.

5.3 Combining Structured with Full-text Queries

The main objective when integrating full-text search into structured queries was to retain the query syntax so that existing query language parsers can be reused. The standardized RDF query language SPARQL is expandable with so called extension functions[9] that can have multiple RDF terms as input parameters and one output value. Unfortunately, such functions cannot be used for our purposes, because they return only one value. Thus, it cannot provide valuable information from full-text search such as the rank of documents or snippets.

Query 1 Example hybrid query expressed in SPARQL.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX ls: <http://www.openrdf.org/contrib/lucenesail#>
SELECT * WHERE {
 ?uri1 rdf:type ?type1 ; rdfs:label ?label1 .
 ?uri2 rdf:type ?type2 ; rdfs:label ?label2 .
 ?uri1 rdfs:seeAlso ?uri2 .

 ?uri1 ls:matches ?q1 .
 ?q1   ls:query "john"  ; ls:predicate rdfs:comment ;
       ls:score ?score1 ; ls:snippet ?snippet1 .

 ?uri2 ls:matches ?q2 .
 ?q2   ls:query "smith" ; ls:predicate rdfs:comment ;
       ls:score ?score2 ; ls:snippet ?snippet2 .
}
```

Another way to extend the query language is to use *virtual properties*. These RDF proper-

[9] SPARQL — Extensible Value Testing: http://www.w3.org/TR/rdf-sparql-query/#extensionFunctions

ties form a distinct full-text query resource inside the structured query, which is actually not materialized in the indexed RDF data. Its purpose is to annotate RDF resources with full-text queries and to provide means to access various information from the full-text search, as opposed to the SPARQL extension functions. Example Query 1 of such a hybrid query is illustrated in Figure 5.2. It queries for two resources (`?uri1` and `?uri2`) that are connected via the `rdfs:seeAlso` predicate. One resource matches the keyword `"john"`, while the other matches `"smith"`. Both full-text queries are restricted to the `rdfs:comment` predicate of the matching resource. Lucene score and snippet information are requested for both full-text queries.

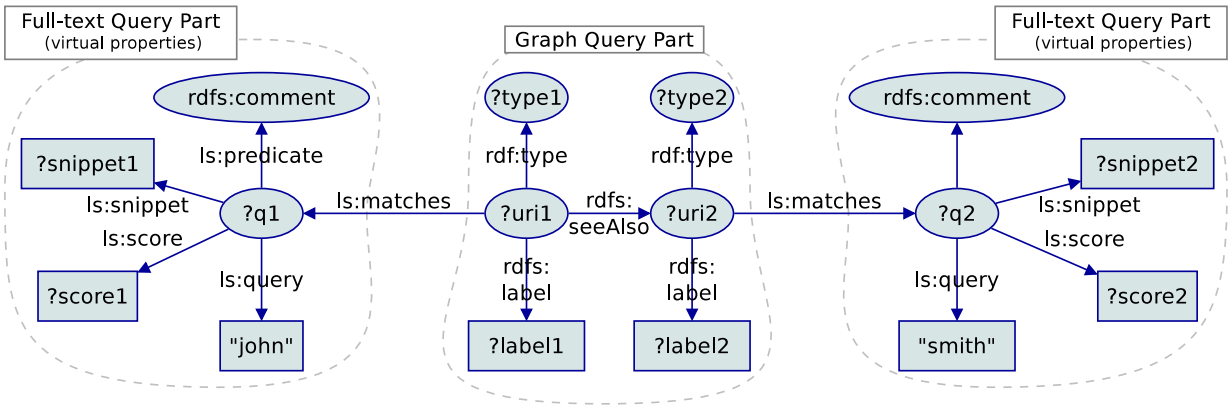


Figure 5.2: An example LuceneSail hybrid query.

In the layered architecture shown in Figure 5.3, the virtual properties are extracted and corresponding actions on the full-text index are performed. The non-virtual part of the query is sent to the underlying layer (the wrapped RDF store) and the results from both parts are joined into the final result that satisfies the full hybrid query. The main advantage of this two-layer approach is that existing RDF stores can be used and transparently extended with full-text capabilities. Further, different query languages and existing parsers can be used without modification.

5.4 Integrating Lucene into Sesame: the LuceneSail

We have chosen to implement our full-text search solution as a Sesame Storage And Inference Layer (SAIL) because this allows for easy extension of existing SAILs with our full-text search.

This approach has several advantages: Firstly, the LuceneSail is independent of any underlying RDF storage. It works equally well on top of the Sesame NativeRDF store as on a relational database. Secondly, our LuceneSail will be query-format agnostic, since we only hook into the query model provided by Sesame's query parser. The LuceneSail has two modes of operation within Sesame: the initial indexing of RDF data and handling virtual properties in queries.

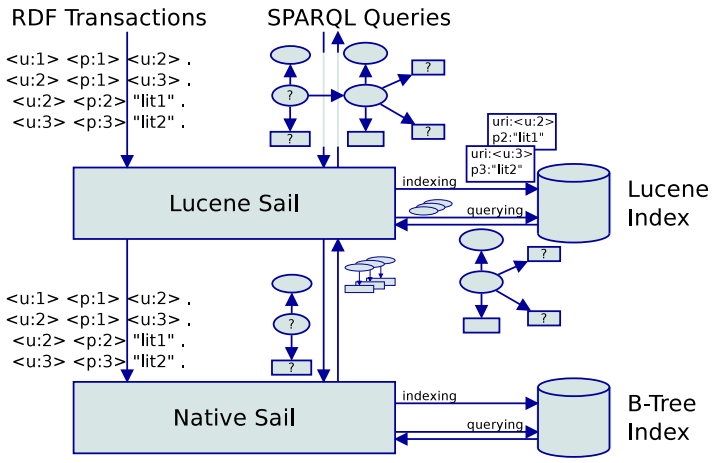


Figure 5.3: Layered architecture of the full-text extension.

5.4.1 Storing RDF in Lucene Documents

Storing RDF in Lucene documents needs a mapping from the graph-structure of RDF to Lucene documents. As described above, such a document contains name-value fields. For full-text indexing of RDF data we identified two possible strategies for mapping RDF to Lucene documents:

Statement-based Each RDF statement becomes one Lucene document.

Resource-based Each Lucene document represents one RDF resource and all its predicates.

Both approaches have their advantages and disadvantages. In modern RDF stores, it is common to extend the RDF data model with context information. This is used to split a large RDF store into smaller named graphs. In the statement-based approach, this can be achieved by an context field. For the resource-based approach, one Lucene document can be generated for every context of the resource. This introduces overhead of processing multiple Lucene documents on query time, as well as problems computing the relevance score for the merged document.

The second bit of information lost in the resource-based approach is the additional metadata about RDF literals (language and data-type tags). In the statement-based approach, these can be added as additional fields. Here, it is also problematic to query for a keyword occurring in any of the predicates. A joined query over all fields would not scale due to the potentially high number of RDF predicates (and therefore fields). An additional field that stores the content of all fields has to be updated whenever new data arrive or get deleted.

However, the representational power of the statement-based approach comes at the cost of having to index many more Lucene documents. Furthermore, in order to query for statements that have a specific predicate and literal matching a given keyword, two indices must be queried and joined. Also, querying for resources that match in multiple specific predicates cannot be merged by Lucene directly, as it can with the resource-based approach. And finally, since all literals are indexed in the same inverted index (field “literal”), there is only one global score over all literals, and not a predicate-based relevance score.

Join Tree 1 Query plan of the example query as a Join Tree.

```
SELECT ?uri1, ?score1, ?snippet1, ?uri2, ?score2, ?snippet2,
       ?label1, ?type1, ?label2, ?type2
  JOIN
    JOIN
      JOIN
        JOIN
          JOIN
            JOIN
              JOIN
                JOIN
                  JOIN
                    JOIN
                      JOIN
                        JOIN
                          JOIN
                            JOIN
                              JOIN
                                ?uri1 rdf:type ?type1                    (SP1)
                                ?uri1 rdfs:label ?label1                 (SP2)
                              ?uri2 rdf:type ?type2                      (SP3)
                            ?uri2 rdfs:label ?label2                     (SP4)
                          ?uri1 rdfs:seeAlso ?uri2                       (SP5)
                        ?uri1 ls:matches ?q1                             (SP6)
                      ?q1 ls:query "john"                                (SP7)
                    ?q1 ls:predicate rdfs:comment                        (SP8)
                  ?q1 ls:score ?score1                                   (SP9)
                ?q1 ls:snippet ?snippet1                                 (SP10)
              ?uri2 ls:matches ?q2                                       (SP11)
            ?q2 ls:query "smith"                                         (SP12)
          ?q2 ls:predicate rdfs:comment                                  (SP13)
        ?q2 ls:score ?score2                                             (SP14)
      ?q2 ls:snippet ?snippet2                                           (SP15)
```

We decided to use the resource-based approach, despite of the reduced representational power. We evaluated that the context information is not essential for implementing useful full-text search. An early version of the LuceneSail has been deployed for the Semantic Desktop search engine *Beagle*$^{++}$ [122], the Semantic Desktop prototype Gnowsis [165], and in the NEPOMUK Project [131], where this feature was never required. The missing language and data type tags are hard to support, however, they always can be retrieved from the underlying RDF store.

According to the resource-based approach, LuceneSail generates Lucene documents with one field that holds the URI (field name `uri`) of the respective resource, one field with all literals' content (field name `all`), and one field for each literal with the predicate as the field name. In a sense, this Lucene document generation from RDF statements is the inverse function of the Lucene to RDF translation performed in Section 3.4.5.

Since Lucene has no concept for updates, we implement updates by deleting and adding an updated version. To allow reconstructing the `all` field, it has to be stored. Alternatively, the underlying RDF store could be queried for this information, but this would incur performance penalty. Another option is to create separate Lucene documents and merge them on query time.

5.4.2 Querying the LuceneSail

Join Tree 2 Reduced query plan as sent to the underlying SAIL with Lucene result bindings.

```
        ?uri1, ?score1, ?snippet1,
SELECT  ?uri2, ?score2, ?snippet2,
        ?label1, ?type1, ?label2, ?type2

JOIN
  JOIN
    JOIN
      JOIN
        JOIN
          ?uri1 rdfs:seeAlso ?uri2                    (SP5)
          ?uri1 rdf:type ?type1                       (SP1)
        ?uri1 rdfs:label ?label1                      (SP2)
      ?uri2 rdf:type ?type2                           (SP3)
    ?uri2 rdfs:label ?label2                          (SP4)
```

with specific bindings on ?uri1, ?score1, ?snippet1, ?uri2, ?score2, ?snippet2

The LuceneSail processes a hybrid query in four phases. Initially, the query is split into the full-text and the graph query parts. The full-text query part can contain multiple full-text queries referring to distinct resources. If there is no full-text query part, the query is completely evaluated by the underlying SAIL. Otherwise, the full-text queries are evaluated using the Lucene index. For each matching resource, the respective variable (`?uri1` and `?uri2` in Figure 5.2) in the graph query part is bound. It is then passed to the underlying SAIL for evaluation. The LuceneSail also binds the optional score and snippet variables.

In case all variables are bound, the evaluation by the underlying SAIL degrades to an existence operation: *Does this binding exist in the data store?* If there are still unbound variables, the graph query part is a valid query and results from the SAIL are partial results of the hybrid query.

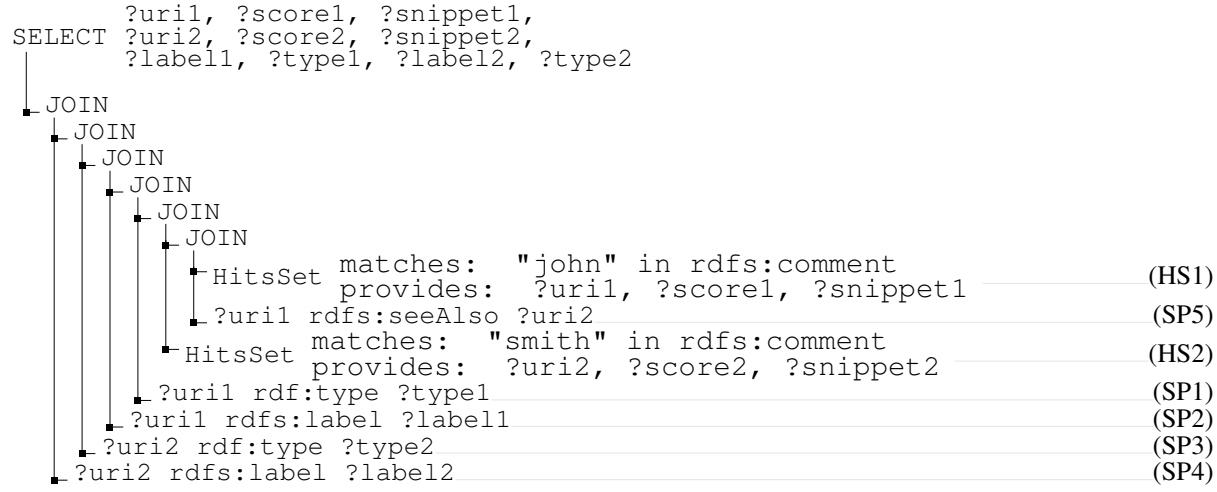
This approach can be seen as a join operation of the Lucene results with the results of the graph query part. Lucene results are looked up by the underlying SAIL if they exhibit the respective structural properties. When multiple full-text queries exist, they are all joined to get their Cartesian product and used as detailed above.

This join-by-variable-binding proved to be an effective integration approach since no modification of Sesame was necessary. The full-text extension is transparent to the underlying SAIL. However, this integration only performs optimally when the cardinality of full-text search result and consequently the Cartesian product of the results are smaller than the cardinality of the graph query part. For some semantic applications, *e. g.*, *Beagle*$^{++}$, multiple full-text queries are not needed. Then, this lightweight integration is sufficient and efficient. If multiple full-text queries are required, as for instance to evaluate queries constructed with the NEPOMUK Structured Query Builder (see Chapter 4) or our QUICK system [192], a tighter integration is needed.

Set-Injection Integration into Sesame For semantic applications that require multiple full-text queries within one hybrid query, we further developed a tighter integration of LuceneSail into Sesame. The core idea of our improved LuceneSail integration is to inject a special operation node into the query evaluation plan (the Join Tree) that represents a whole full-text query part.

For easy integration of this operation, we introduced the new generic `Set` node, which is an abstract operation node that provides a fixed set of result bindings for particular variables. The LuceneSail then provides its particular `LuceneHitsSet` implementation, which represents all matching resources provided as bindings of a particular full-text query. This integration requires some modifications that are not available from the Sesame sources and therefore demand a patched version of Sesame, which is publicly available[10].

Join Tree 3 Reduced query plan of the example query using HitsSet.



Join Tree 1 depicts the query plan of the example query from Figure 5.2. In the Join Tree 3, all full-text search relevant statement patterns (`SP6`-`SP10` and `SP11`-`SP15`) are replaced by the HitsSets `HS1` and `HS2`. Each HitsSet provides bindings for the matching resource (`?uri1`, `?uri2`), the Lucene score (`?score1`, `?score2`), and the snippets (`?snippet1`, `?snippet2`).

In RDF stores, the join operator is usually implemented as the *Index Nested Loop Join* [34,91]). This join is evaluated by executing its first (the upper in the query plan examples) argument using the currently existing bindings. Then, each result is appended to the current bindings and used to evaluate the second (the lower) join argument. In our example Join Tree 3, the HitsSet `HS1` is evaluated first. Each result provides a binding for `?uri1`. For each such binding, the evaluation of the statement pattern `SP5` provides bindings for `?uri2`. Then, the `HS2` is evaluated with a given binding for `?uri2`. This evaluation degrates to a boolean query: *Does resource `?uri2` match the keyword?* In order to improve the performance of this look-up, we pre-fetch all `?uri2` that match the query of `HS2` (if the result set is below a certain threshold). Here, we trade repetitive Lucene look-ups that cause expensive I/O operations for increased memory consumption.

Finally, our HitsSet provides cardinality information to the query planner. The cardinality is exactly known due to the pre-fetching described above. When a binding for the resource variable is provided (`?uri1`, `?uri2`) there can be at most one result. Therefore, the cardinality

[10]The Sesame2 LuceneSail Hits Set flavor: `http://dev.nepomuk.semanticdesktop.org/wiki/LuceneSailFlavorHitsSet`.

is estimated as one. With this information, the query planner can optimize the query plan by freely moving nodes up and down in the query plan to minimize the look-up and iteration costs (see Section 5.5).

Let's continue our running example. All statement patterns have a very high cardinality, because they have no bindings for their subjects and objects. The query planner would evaluate the HitsSet with the lowest cardinality first. Then, statement pattern `SP5` has a binding that probably provides a small cardinality. It will be evaluated next. With the results, `HS2`'s cardinality drops to one and will be evaluated next.

Alternatively, if both HitsSet provide only a small number of results, both HitsSet might be evaluated in the first join operation, providing the Cartesian product of their results. Then, the statement pattern `SP5` has bindings for `?uri1` and `?uri2`, which is a simple existence look-up: *Is resource `?uri1` connected to `?uri2`?* In data sets where the general connectivity of resources is high, this strategy could be the optimal one. Our query plan optimization algorithm (see Section 5.5.3) will be able to construct the optimal plan also in such a case.

5.5 Improving Query Performance of Sesame

We have seen that hybrid queries require an efficient query evaluation to guarantee low response times of semantic applications. Existing stores are particularly challenged by complex hybrid queries. As a first improvement to tackle this issue, I present a general cardinality estimation for on-disk B-trees, as they are employed by many RDF stores. With this information, a more sophisticated join ordering strategy can be applied. I report on a Greedy strategy that searches the space of possible query plans and is more likely able to construct a more efficient query plan. Further, the evaluation of RDF queries usually requires a large number of range queries [75, Sec. 13.3.4] that invoke index look-ups and iterations. I present an on-disk B-tree optimization that improves performance of these frequent operations. Finally, a short analysis on the impact of index selection for a particular range query is discussed that emphasizes the need to consider the dynamics of this task. In short, my improvements target these aspects of query evaluation:

- Better cardinality estimation and statistics
- Join ordering strategy
- Optimization of on-disk B-trees
- A better index selection strategy

In the following, I provide details on these improvements done to Sesame's query evaluation. The performance evaluation in Section 5.6 reports on achieved improvements over the original Sesame implementation. Chapter 6 on page 103 compares the improved Sesame implementation with the performance of three other mature RDF stores by an in-depth performance analysis.

5.5.1 Better Cardinality Estimation

In the following, this notation is used to refer to particular kinds of statements:

- `p` - constants like a URI or a Literal
- `?s` - unbound variable
- `!o` - bound variable

Let (`s p o c`) be a statement that consists of subject `s`, predicate `p`, object `o`, and context `c`. In the following, for the sake of simplicity and without loss of generality, we assume the context `c` to be constant and therefore omit its value from our notation.

The *NativeRDF* SAIL of Sesame, as many other native RDF data stores [37, 89, 91], employs on-disk B-tree data structures to index statements (`s p o`). Such an index organizes the statements in a particular order. Each value of `s`, `p`, and `o` is mapped to a unique integer value. The statement order `pos`, for instance, sorts all statements based on the integer representation of property `p`. Statements with identical predicates are further sorted by their object `o`, and finally by the subject `s`. Under the assumption that the predicate `p` is always specified in a statement pattern, we can categorize statement patterns into four types. For each of these types we use a different method to estimate the cardinality.

(s p ?o) For patterns of this type, where `s` is specified and `?o` is an unbound variable, we can exploit the `spo` B-tree index to estimate the cardinality (see below).

(?s p o) The cardinality is estimated similarly to (`s p ?o`) patterns using the `ops` index.

(!s p ?o) This type of patterns handles the case when variable `?s` is bound by a previously evaluated operation. The distinction to the `(s p ?o)` case where `s` is a specific value is important since here, `!s` is a bound variable. The actual value of `!s` is not known in the query planning phase, but it is known that it will be given in the evaluation phase. Therefore, the B-tree index cannot be employed to estimate the cardinality of the unknown value of `!s`. Here, statistics on the general cardinality of any `s` in conjunction with the specific predicate `p` are employed. Assuming a uniform distribution of values of `?s` among all statements $\mathbf{stms}_p \in \mathbf{D}$ of the RDF data set $\mathbf{D}$ that contain predicate `p`, the cardinality of this type of pattern can be estimated as $card_{s,p} = |\mathbf{stms}_p| \,/\, |\mathbf{subjects}_p|$, where $\mathbf{subjects}_p$ are the distinct subjects that occur in these statements.

Let's consider these examples. The pattern (`?uri rdfs:label ?label`) has a cardinality of one if every resource has assigned exactly one label. In contrast, the pattern (`?uri rdf:type ?type`) has a cardinality larger than one in a data set that has a schema with a large sub-class-hierarchy and full transitive closure (see Section 2.1). Then, a resource of a particular type is also of all its super classes' types, which is at least one. Though these two instances of patterns are of the same type, they expose different cardinalities.

(?s p !o) The same technique as for (`!s p ?o`) is applied, replacing `s` by `o`.

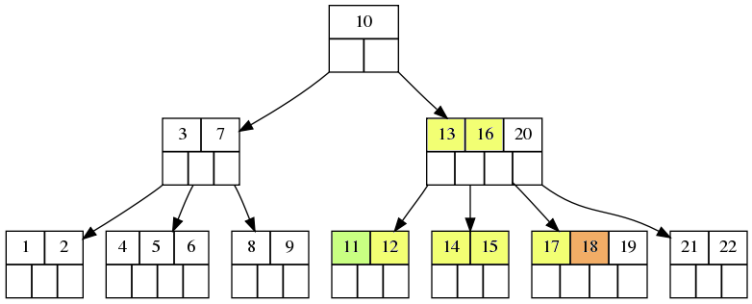


Figure 5.4: B-tree range query cardinality estimation: Smallest statement #11 (green), largest statement #18 (red), isolating the result set (#11–#18) of a certain statement pattern.

(!s p !o) Cardinality is estimated as one, since this is an existence look-up and is either true or false. Considering a setup where statements exist with a number of contexts, the cardinality of this statement pattern should be the average number of contexts a statement is valid for.

For statement patterns where a subset of `s`, `p`, and `o` is specified, a B-tree can quickly provide all statements with the given values via a *range query* [75, Sec. 13.3.4]. Given a `pos` index and a statement pattern (`?s rdf:type rdf:Resource`), a look-up for the smallest possible statement with the integer value for `s` set to 0 returns the smallest matching statement (`minStatement`). A look-up for the largest possible statement with `s` set to the largest integer value returns `maxStatement`. Based on the path through the B-tree that is taken to reach those two statements (`minPath` and `maxPath`), it is possible to give a good estimation of the number of results that are within the range of these two statements.

The `minPath` and `maxPath` values for the example shown in Figure 5.4 are

- `minPath` : $([1,2],[0,4],[0,2])$
- `maxPath` : $([1,2],[2,4],[1,3])$

where a path is a sequence of steps denoted as $[idx, N]$, idx reflects the zero-based index of the child followed and N refers to the number of children in the current node. The first step refers to the root node. The last step in the path sequence always refers to the node containing the statement. There, idx denotes the statement's index and N refers to the number of statements in that node.

Let's see how the number of statements that are contained in the range spanned between `minPath` and `maxPath` can be estimated from these paths. At the root level, both paths follow the same child ($idx = 1$), so no cardinality is indicated here. At the second level, the paths split and have one child node in between (the child node with statements #14 and #15). In order to avoid examination of the middle child tree, its cardinality is estimated based on its height and the expected fill rate. In literature, the expected fill rate after random insertion is reported to be $\ln 2 \approx 69.3\,\%$[11] [110, p. 489]. The current level further adds two statements to the estima-

[11]Note that with our optimized on-disk B-tree, the fill size is more accurately known (see Section 5.5.4).

tion (#13 and #16). At the last level, the `minPath` contributes the statements #11 and #12, the `maxPath` contributes the statements #17 and #18. This estimation does not require fetching any B-tree nodes in addition to the `minPath` and `maxPath` look-up.

This example shows that for all nodes on the two paths, the cardinality is accurately given by the path sequence. For all sub-trees that span to the right of the `minPath` and to the left of the `maxPath` can be estimated based on their height. This shows that the smaller the real cardinality is, the more accurate our approach is. Recall that the estimated cardinalities are later on compared, so in the end, the correct order of the statement patterns (ordered by estimated cardinality) is more important than the accuracy of the estimated values.

The presented B-tree cardinality estimation was integrated into the Sesame sources[12].

Complexity

The described cardinality estimation approach requires at most $|\texttt{minPath}| + |\texttt{maxPath}| - 1$ I/O operations, where $|\texttt{minPath}|$ and $|\texttt{maxPath}|$ denote the number of steps of the respective sequence, which is at most $\lceil log_B(N+1) \rceil - 1$, where B refers to the branching factor of the B-tree and N the number of entries in the tree. Due to caching, all nodes that both paths have in common might be read only once, which further reduces the number of I/O operations.

5.5.2 Gathering Cardinality Statistics

As discussed before, as soon as variable bindings are available for statement patterns, statistical information is needed for cardinality estimation. We provide such statistical information about statement patterns that have a given predicate, which is usually the case.

So we saw that with the number of statements having a certain property $\mathtt{p}$ ($\mathbf{stms}_p$), the number of distinct subjects ($\mathbf{subjects}_p$) and objects ($\mathbf{objects}_p$) among these statements, those statistics become available. Maintaining a counter and a set of subjects and objects for each property as new statements are added to the RDF store is trivial. The sets can be handled by a B-tree for each property. Existing values will be ignored when added to a B-tree. The tree therefore only contains distinct subjects or objects, and its size provides the exact size of the respective set.

The extraction of those numbers is also possible in an offline fashion and scales with the number of properties and statements. Given a `pos` index, one iteration over this index is sufficient to extract the number of distinct objects for each property, with a constant memory consumption of two counters, the current statement and the last property and object, as sketched in Algorithm 1. The core idea of that algorithm is to exploit the existing order of the statements. This order aggregates all statements by the property. When a transition of the property in the stream of statements from $p\prime$ to p occurs, the property $p\prime$ will never appear again. Therefore, the exact cardinality of statements with property $p\prime$ is known after such a transition. The same effect can be exploited to count the distinct objects for a particular property. All objects co-occurring with a particular property are grouped into one continuous sequence. After a transition from object

[12] Aduna WebSVN: `http://repo.aduna-software.org/svn/org.openrdf/sesame/trunk/core/sail/nativerdf/`. See `getValueCountEstimate(byte[], byte[])` method in `org.openrdf.sail.nativerdf.btree.BTree` in revision 7601 on August 15, 2008.

Algorithm 1: Cardinality Statistics Algorithm `cardStatistics` for `pos`

Input: iterator it over statements ordered by `pos`
Output: sequence $\mathbf{S}$ of property cardinality $card_p$ and property-object cardinality $card_{p,o}$ represented as tuple $(p, o, card_p, card_{p,o})$

```
// initialize current property and object,
// statements and distinct objects counters
(p', o', N_s, N_o) ← (null, null, 0, 0) ;
S = ∅ ;                                          // reset statistics
while it has more statements do                  // iterate over statements
    (s, p, o) ← it.next ;                        // get next statement
    if p ≠ p' then                               // p is a new property
        if o' ≠ null then
            S ← S ∪ {(p', N_s, N_o)} ;           // store statistics of property p'
        (p', o', N_s) ← (p, null, 0) ;           // reset statistics for new p'
    if o ≠ o' then                               // o is a new distinct object
        o' ← o ;                                 // memorize new object o
        N_o ← N_o + 1 ;                          // count new distinct object
    N_s ← N_s + 1 ;                              // count statement with property p'
if o' ≠ null then
    S ← S ∪ {(p', N_s, N_o)} ;                   // store statistics of property p'
return S
```

$o\prime$ to o, while the same property remains, we know an object o was never observed before and is therefore a distinct object regarding the current property.

Another full iteration over the `pso` index provides the number of distinct subjects for each property. Under the assumption that minimal changes of the RDF data do not significantly change those statistics, it is sufficient to gather these statistics offline on an infrequent basis.

Complexity

Our offline strategy to gather cardinality statistics requires one full iteration of the `pos` and `pso` indices, which is an I/O complexity of $O(N)$ with N being the number of statements. Further, it requires a constant amount of memory $O(1)$ and a disk storage complexity of $O(P)$ with P referring to the number of distinct properties. Access to the statistics can be achieved via in-memory data structures that yield to $O(P)$ complexity in size and $O(logP)$ or even $O(1)$ in time using binary search or hash maps, respectively.

5.5.3 Approximation of the optimal Join Tree

Cardinality estimation for unbound and bound patterns allows for a detailed I/O cost estimation of a given join tree [75, Sec. 16.4]. The following values are of interest for this estimation:

- look-up cost $l(sp)$ — the I/O cost to look up statement pattern sp (retrieve the first result)
- iteration cost $it(sp)$ — the I/O cost to iterate over all results of statement pattern sp
- cost $c(sp)$ — the total cost of evaluating one statement pattern is $l(sp) + it(sp)$
- cardinality $|sp|$ — the cardinality of the result set of statement pattern sp

When statements are stored in a B-tree, $l(sp)$ is proportional to the height of the B-tree and iteration cost $it(sp)$ is proportional to the cardinality $|sp|$[13].

A join operation *left* $\bowtie$ *right* creates the set of those results in the Cartesian product that satisfy the join constraint. The most important join algorithms are the *index nested loop join*, the *zig-zag join*, and the *hash join*.

The *index nested loop join* $\bowtie_{inl}$ [75, Sec. 15.6.3] looks up one argument (here the left argument *left*), iterates over the result set and looks up its other argument (the right one *right*) using the current result of the left argument as variable bindings. Therefore, a nested loop join has the following properties:

$$l(\textit{left} \bowtie_{inl} \textit{right}) = l(\textit{left}) + |\textit{left}| \cdot l(\textit{right}) \tag{5.3}$$

$$it(\textit{left} \bowtie_{inl} \textit{right}) = it(\textit{left}) + |\textit{left}| \cdot it(\textit{right}') \tag{5.4}$$

$$c(\textit{left} \bowtie_{inl} \textit{right}) = l(\textit{left}) + it(\textit{left}) + |\textit{left}| \cdot (l(\textit{right}) + it(\textit{right}')) \tag{5.5}$$

$$0 \leq |\textit{left} \bowtie_{inl} \textit{right}| \leq |\textit{left}| \cdot |\textit{right}| \tag{5.6}$$

Note that the right argument is evaluated using the left argument's results as variable bindings, denoted as *right*′, which usually has a smaller cardinality than without bindings.

The *zig-zag join* $\bowtie_{zz}$ [75, Sec. 15.6.4] has the prerequisite that the result set of its arguments are ordered. This can be expected in some cases since the `spo` index, for instance, provides results for a given subject ordered by predicate, object and context. If this useful order [172] is not provided, the arguments have to be sorted in-memory or on-disk first. This adds additional I/O and time to the cost calculation, which can easily be integrated into the join cost calculation.

Given ordered results, a zig-zag join has the following properties:

$$l(\textit{left} \bowtie_{zz} \textit{right}) = l(\textit{left}) + l(\textit{right}) \tag{5.7}$$

$$it(\textit{left} \bowtie_{zz} \textit{right}) = it(\textit{left}) + it(\textit{right}) \tag{5.8}$$

[13] In Sesame 2.2, $l(sp) = 5$ for 1×10^9 statements and $it(sp) \approx |sp| \times 8$

The *hash join* $\bowtie_h$ [75, Sec. 15.5.5] loads the entire result set of its right argument into memory and then performs the evaluation using the variable bindings from the left argument purely in memory. It has the same properties as the zig-zag join, except that there is no prerequisite of a specific order of both arguments' result sets. Further, it has a much larger memory consumption, which is proportional to $card(\textit{right})$. This shows that memory consumption should also be considered in the cost estimation in order to avoid selecting this algorithm for joins that have a large result set on the unbound right argument.

Join Order Optimization: Exploring the Space of Join Trees

Among the set of SPARQL algebra operators (*select*, *join*, and *statement pattern*), in most cases query execution costs are dominated by the order of *join* operations. Any permutation of a join tree's nodes produces the same result set but at different costs. This makes finding the optimal join tree crucial for efficient query processing. In this work, we restrict our optimization to left-deep join trees [75, Sec. 16.6.3] (see Join Tree 1). Every inner node in such a tree is a join operation where the left child is a left-deep join tree itself providing variable bindings to be joined with the single right child operation. Such a join tree can be stated as a sequence of leaf operations in order of their evaluation [140]: $(s_1, s_2, \ldots, s_{n-1}, s_n) \Leftrightarrow ((\ldots((s_1 \bowtie s_2) \bowtie s_3)\ldots) \bowtie s_{n-1}) \bowtie s_n$

For a query having N statement patterns to be joined, there are $N!$ possible join trees. Further, each join operation could be performed by one of the three different join algorithms introduced above, which further increases the permutation space. An exhaustive generation and test of all permutations is too expensive even for small N. Approximation algorithms like dynamic programming are used to identify a near-optimal join order [172].

We start with N join trees, everyone consisting of one of the available statement patterns. Each join tree furthermore has a list of statement patterns associated that are not yet considered for exploration (non-joined), initialized with the other $N - 1$ statement patterns. Each join tree has its cost computed. As long as the cheapest join tree does not contain all input statement patterns, a new join tree is generated by joining it with the non-joined statement pattern with smallest cardinality[14]. For the originating join tree, the respective statement pattern is then removed from the list of unexplored statement patterns to prevent the creation of duplicate plans. The new join tree has all non-joined input statement patterns in its non-joined list, since it constitutes the root for a new exploration branch. This allows to only continue exploration with the cheapest join tree in every step and join trees that have exploding costs are stalled.

In contrast to Selinger *et al.* [172], our planning exploration also considers full Cartesian products which provides better query plans with controlled search complexity [140]. Cases where Cartesian products have exploding costs are early identified and pruned.

[14]The statement pattern with the smallest cardinality is taken because it is then likely for the new join tree to have a small cardinality, too.

Algorithm 2: Greedy Query Planner

Input: Set of Statement Patterns $\mathbf{S}$
Output: Query Plan (Statement Pattern Sequence) $\mathbf{p}$

```
1  if |S| ≤ 1 then                    // for less than two statement patterns ...
2      return S                        // ... there is a trivial solution
3  P = ⊘ ;                             // initialize the plans to be explored ...
4  for ∀s ∈ S do
5      p = (s) ;                       // ... with one plan ...
6      P ← P ∪ {p} ;                   // ... for each statement pattern
7      f(p) ← S \ {s} ;                // initialize exploration trails for p
8  p* = arg min_{p∈P} c(p) ;           // current cheapest (partial) plan
9  while S \ p* ≠ ⊘ do                 // unless p* is a complete plan ...
10     s* = arg min_{s∈f(p*)} |s| ;    // exploration trail for p* with min. card.
11     p' = (s_1, ..., s_n, s*) with p* = (s_1, ..., s_n) ;   // extend p* by s* to p'
12     P ← P ∪ {p'} ;                  // add p' to plans P to be explored
13     f(p') ← S \ p' ;                // initialize exploration trails for p'
14     f(p*) ← f(p*) \ {s*} ;          // update exploration trails for p*
15     if f(p*) = ⊘ then               // exploration of p* completed?
16         P ← P \ {p*} ;              // remove p* from plans P to be explored
17     p* = arg min_{p∈P} c(p) ;       // update cheapest (partial) plan
18 return p*
```

I integrated this join tree space search in Sesame. Unfortunately, Sesame provides only one join implementation, namely the Nested Loop Join, and we refrained from extending Sesame's architecture to enable multiple join implementations. However, even with this limitation of join algorithms, our join tree planner achieves significant performance improvements. The addition of further join algorithms is considered future work.

Complexity

In best case, the query planner exhibits linear runtime *w. r. t.* the number of joined statement patterns. In contrast, the worst case is still factorial in a situation when each initial one-statement plan is explored along all exploration trails first, which leads to $n(n-1)$ two-statement plans. Continuing this exploration leads to $n!$ plans. However, in practice, this situation should be very unlikely. From the pseudo-code of Algorithm 2 we can observe that the algorithm terminates as soon as all partial plans are more expensive than the cheapest full plan. Since the costs of a plan accumulates with each sequence extension, the cheapest full plan tends to also be cheaper than alternative plans of the same size at intermediate phases. Other dynamic programming approaches have been shown to generate at most $O(N^3)$ plans, thus being polynomial in time [100].

5.5.4 Optimization of on-disk B-trees

B-trees [54, 110] store key-data tuples in a specific order to allow for very efficient retrieval of data given a key. Such a tree is the central index structure of many RDF stores, where in this setting, no external data is associated with keys because the keys are the data, thus statements are indexed *and* stored in B-trees. During search operations, the tree is heavily accessed so that an optimization for frequent query-time operations can significantly improve search performance. Such frequent operations are the look-up of a single key and the iteration over subsequent keys.

Primary characteristic of a B-tree is its branching factor B which defines that nodes of the tree have at most B and at least $B_{min} = \lceil \frac{B}{2} \rceil$ children. An exception is the root node, which may have non or at least two sub-trees[15]. All nodes of a B-tree contain one key less then it has children. All leaf nodes, these are nodes without child trees, reside at the same level. A tree of height h can accommodate at most $N_h = B^{h+1} - 1 : h >= 0$ keys, whereas a tree of height $h = 0$ is a single root node. The minimum number of nodes required to store N keys is $\lceil \frac{N}{B} \rceil$.

For branching factors $B \geq 3$, numerous trees with the same cardinality N exist, denoted as the forest $\mathcal{T}_N^B$ [154]. In this forest, we want to find or rather construct a tree that meets the following criteria. See Figure 5.5 for an example of such an optimal B-tree.

1. The tree has the minimal number of nodes that is required to hold N keys.

2. Nodes are stored on disk in the order of the fetching sequence for a full iteration.

In extension to [154], we specifically consider on-disk B-trees, which also comprises the actual location of pages allocated on the disk. For a fixed B-tree structure, any permutation of allocated disk pages reflects an individual on-disk B-tree in that sense.

In the following, I present a strategy that constructs from a given B-tree a space-optimal B-tree, *i. e.*, a B-tree with minimal number of nodes, a so called compact B-tree [120, 152–154]. Further, the constructed tree allocates disk pages in the order of a full iteration which minimizes seek distances for any iteration over the tree's keys. An input B-tree is used to provide the keys in the desired order of the output B-tree.

In contrast to B+-trees for which such a construction is easily done by creating full nodes and building a minimal tree structure in a bottom-up fashion [21, Sec. 2.3], the construction of a compact B-tree is a non-trivial operation. Simply inserting ordered keys into a B-tree leads to minimum storage utilization [42, 154]. Even with a sequence of keys that results in an optimal tree, the reorganization operations that are required during key addition to properly split nodes [110, Sec. 6.2.4] cause a computational overhead which renders this strategy inefficient.

Efficiency can be achieved by considering the construction of a compact B-tree as a single operation that directly creates the optimal nodes in a holistic process. Such an operation is generally called a bulk load [42], which can be classified into sorting, buffer- and sampling-based bulk load approaches [21]. In this classification, our approach is most similar to sort-based approaches which requires on the keys to be properly sorted.

[15] I use *child* and *sub-tree* synonymously when a distinction between the whole sub-tree and the sub-tree's root node does not change the respective statement, but use child when I specifically refer to the root of a sub-tree.

Existing algorithms for bulk loading B-trees do not consider the order of allocated disk pages. The algorithm in [42] requires a "frontier arrangement" phase since it does not use the number of keys to be loaded as an input. In our settings, this information is always present.

Rosenberg *et al.* provide an algorithm that constructs a compact B-tree out of an ordered sequence of keys. It requires to construct the *detailed profile* of the prospect tree in a first phase. Such a profile contains a histogram of the nodes' cardinality for each non-leaf level, *i. e.*, the number of children they link to (see Figure 5.8 on page 98 for examples of such detailed profiles). In contrast, our algorithm computes the minimal number of child trees for each node and distributes the keys that do not fit into the root node among the sub-trees. This distribution again leads to optimal sub-trees. Then, depth-first recursion constructs the entire tree and creates new nodes on disk in the desired sequence. Therefore, our algorithm combines the profile generation and node creation phase in one phase. At any point in time it holds at most $log_B N$ incomplete nodes in memory and it creates new nodes in the order of a full B-tree iteration, which is not considered by the authors. Further, we claim that our algorithm is simpler and more intuitive.

Certainly, such a reconstruction is a costly operation that cannot be performed after each B-tree modification. Rosenberg and Snyder argue that for a moderate number of augmented inserts, the advantages of a compact B-tree "can still be felt". In response to their 1981 publication, Jürgen Klonk disagrees with this feeling and notes that in practice, space-optimality is extremely volatile and maintaining costs are very high [109]. However, as Rosenberg and Snyder, we consider scenarios in which insert operations occur very infrequent or as a batch operation, while reading access represents the majority type of B-tree operations. Then, such a reorganization can be done at the same time as regular backups may be performed and does not cause significant overhead in practical situations.

Compact on-disk B-trees: We name our optimal B-tree a *compact on-disk B-tree*, as it is an extension of a compact B-tree [154] in such a way that nodes are stored on disk in the order of a full iteration over the keys. This particular property provides useful characteristics such as minimized I/O seek operations for iteration operations, as we will see later. Figure 5.5 exemplifies a compact on-disk B-tree with branching factor $B = 4$ and cardinality $N = 32$ keys and Figure 5.6 illustrates compact on-disk B-trees of increasing cardinality ($N = 24, 25, 26$).

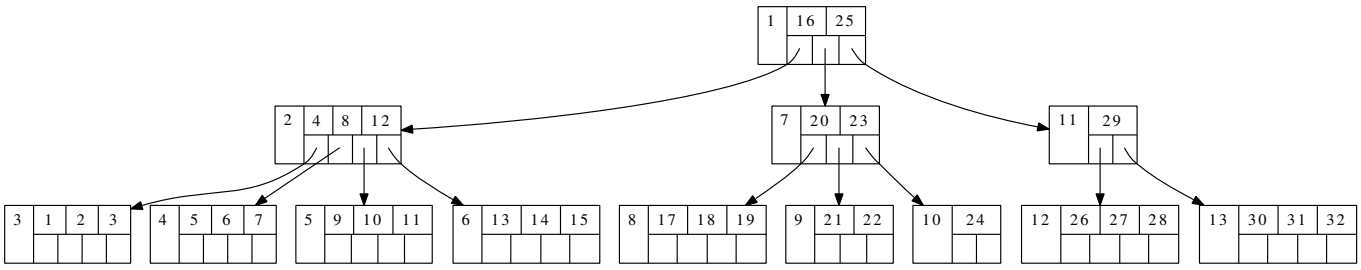


Figure 5.5: A compact on-disk B-tree for branching factor $B = 4$ and $N = 32$ keys. The leftmost number in a node denotes the on-disk position of the page that stores the node, while the other numbers depict the keys.

The following properties of compact on-disk B-trees follow from the two objectives:

The tree is minimal in height: With a minimum number of nodes, the B-tree is minimal in height. In other words, the smallest tree in terms of keys for a given height cannot be made smaller in height with more nodes because this would violate the lower bound of children per node B_{min}. With minimal height, also the path length to the leaf nodes is minimal.

The number of sub-trees of each node is minimized: Starting from the B-tree with minimum number of sub-trees, each additional sub-tree would add nodes to the tree, violating the minimal number of nodes property.

The average keys per node is maximized: B-tree nodes have a fixed size on disk with a fixed number of maximal keys per node[16]. Minimizing the number of nodes when the number of keys contained in the whole B-tree is fixed maximizes the average number of keys per node. Therefore, a compact B-tree provides more keys per fetched and cached node and thus increases fetching efficiency and cache hit probability.

The look-up costs are virtually minimal: It is known that space-optimal B-trees virtually have the same node-visit costs (*i. e.*, the expected number of visited nodes to look up any key) as the visit-optimal tree [154, p. 185]. Thus, key look-up is practically optimal.

The sum of absolute I/O seek distances are minimized for iteration: Assume that all h nodes that are to be fetched to find a key in a leaf node are hold in memory [110, Sec. 6.2.4]. Then, for an iteration operation as a range query, the distances from any currently visited node to the next un-cached node is always one node size. Looking at the example depicted in Figure 5.5, an iteration from key 15 to key 26 causes six seek operations all with distance of one node size: from node 6 containing key 15 via node 7, 8, 9, 10 and 11 to node 12. Node 1 and 2 are hold in memory since they are on the path to key 15.

Here, the sum distance is the number of fetched nodes times the node size and therefore minimal. Further performance improvements are caused by the fact that subsequent nodes are very likely to be pre-fetched by hard disk or operating system caches.

An optimal B-tree degrades insert operation performance: A compact B-tree has improved performance of read operations, but degrades write operations. Inserting new RDF statements is slower since the creation of new nodes is very likely. In that case, the B-tree is not optimal anymore. Therefore, an optimization is only useful in a scenario where a large batch of RDF statements is indexed and only few write accesses are performed afterwards.

For the construction of such a compact on-disk B-tree we firstly define two trivial cases which frequently occur and which allow us to elegantly define such a tree recursively:

A *full tree* of height h is a tree that exclusively contains full nodes, *i. e.*, nodes with $B-1$ keys. Such a tree has $N_h = B^{h+1} - 1 : h >= 0$ keys and is optimal.

[16] I ignore node compression for the sake of simplicity here.

A *minimal full tree* is defined as a B-tree with a node of minimal size (B_{min} children and $B_{min} - 1$ keys) as its root with full trees as its sub-trees. Consequently, such a tree has $N_{min_h} = B_{min} - 1 + B_{min} \cdot N_{h-1}$ nodes. Obviously, minimal full trees are optimal, since there exists no other tree with the same number of keys but fewer nodes: no key can be moved from any of the full nodes anywhere else without enforcing new nodes.

A compact on-disk B-tree with branching factor B can be recursively constructed by the following strategy. Given the number of keys N to be put into an optimal tree, the minimal height h of that tree is given by $h = \lceil log_B(N+1) \rceil - 1$. A minimal number of root node keys (property one) can be constructed for that tree by first dedicating the maximal number of keys to the left-most child. The remaining keys are distributed among (a minimum number of) additional sub-trees of maximal size. A sub-tree of the root node is of height $h-1$ and therefore stores at most $N_{h-1} = B^h - 1$ keys. For each additional sub-tree, one key has to be added to the root node as well. Then, the minimal number of root keys is given by $N_r = \left\lceil \frac{N - N_{h-1}}{N_{h-1}+1} \right\rceil$, where the dividend represents the keys to be distributed among the $B-1$ children and the root node, and the divisor reflects the number of keys that are consumed by adding one sub-tree. Consequently, the root node has $C = N_r + 1$ sub-trees, where all sub-trees together contain $N_C = N - N_r = N - C + 1$ keys. Constrained by C and N_C, we can construct a set of sub-trees that is optimal (minimal) in the number of its nodes as follows (required by objective two).

A sequence of n full, m minimal full, and one additional sub-tree that accommodates the remaining keys N_{c_r} with $n, m \geq 0$, $n + m + 1 = C$ and $n \cdot N_{h-1} + m \cdot N_{min_{h-1}} + N_{c_r} = N_C$ is optimal when the number of full sub-trees n is maximized. This can be proven as follows. The full sub-trees clearly contain the maximal number of keys they can contain. The same holds for all sub-trees of the minimal full sub-trees. Adding a key to either of the full and minimal full sub-trees causes the creation of additional nodes. Thus, all keys in those sub-trees are stored with a minimal number of nodes. When the remaining keys are also stored in a sub-tree that is minimal, this whole tree must be optimal. The remaining keys can optimally be arranged using this strategy recursively. The recursion terminates when all keys fit into one node, a leaf node.

The minimum full trees are motivated by the observation that a root node splits into two nodes when a key is added to a full tree (see Figure 5.6 for an example). These nodes have $\left\lfloor \frac{B}{2} \right\rfloor$ and $\left\lceil \frac{B}{2} \right\rceil$ keys. One key is moved up to the parent of this tree, or if no parent exists, a new root node with one key is created. In order to still be compact, the first node constitutes the root of a minimal full sub-tree and the second sub-tree accommodates the remaining keys. That tree is filled up with new keys until it becomes a full tree. Then, the first sub-tree becomes the one that grows with new keys. This is the fundamental way how a compact on-disk B-tree grows as an increasing number of keys is used to create it. The remaining keys are always arranged in an optimal way. Therefore, this sub-tree grows in the same way.

A compact on-disk B-tree is constructed by Algorithm 3 `optBTree` following this strategy. The keys sequence K is consumed in one iteration. In contrast to the algorithm in [154], this algorithm allocates pages on the disk in the order of a full iteration (optimality criterion two). Algorithm 4 `optChildren` finds the optimal sequence of sub-trees (more precisely their sizes) by maximizing the number of full sub-trees under the constraint of a minimal number of sub-trees (C) while at most one sub-tree may not be full or minimal full.

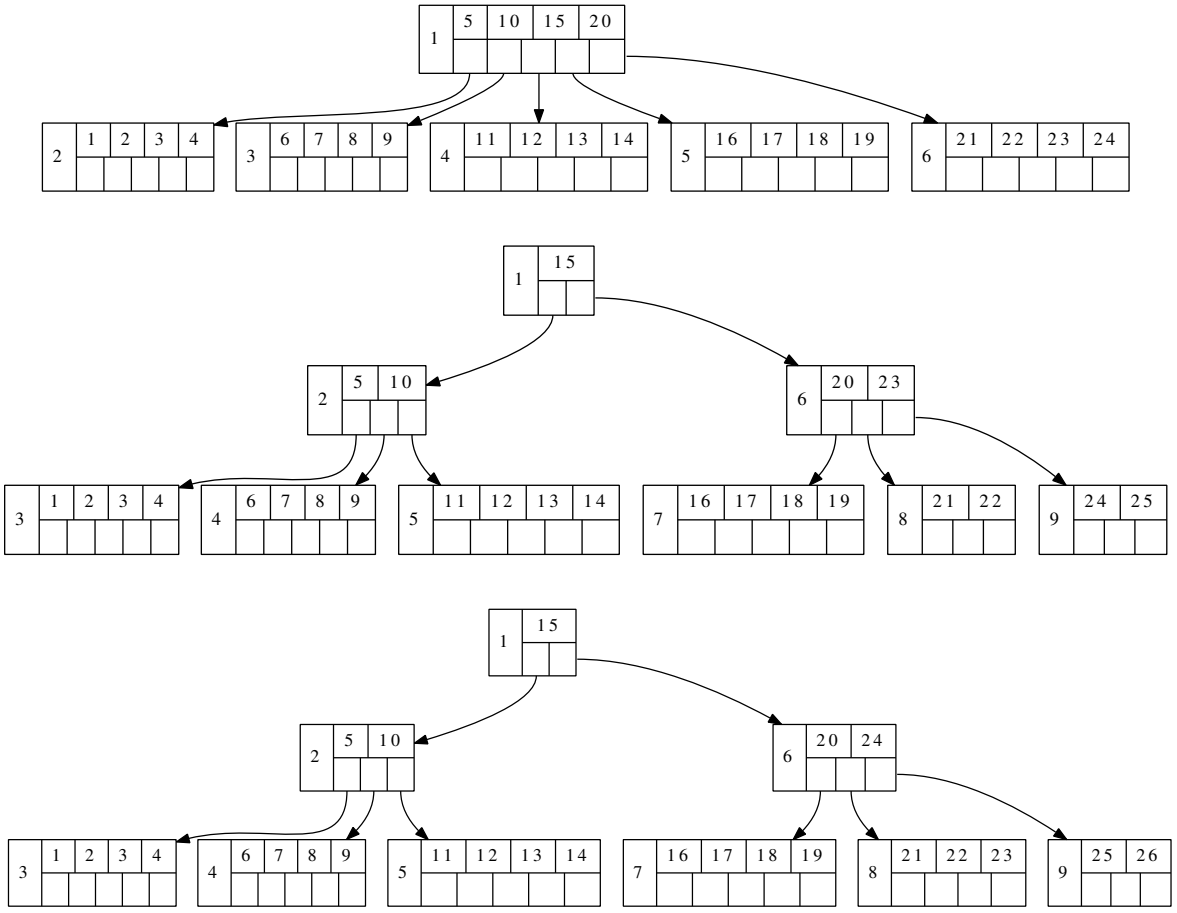


Figure 5.6: A growing optimal B-tree for branching factor $B = 4$ and $N = 24, 25, 26$ keys.

Another useful property of this optimal B-tree can be achieved by sorting the sequence of sub-trees in descending order by their size. Then, the tree becomes left-weighted, *i. e.*, the left children tend to contain more keys than right children. In other words, the left child trees are more likely to contain a certain key than the right ones. Since the I/O seek distances from root node to left sub-trees are smaller then to the right ones, this ordering results in a higher probability for smaller I/O seek distances. On average, the I/O seek distances are lower with this ordering.

I integrated the B-tree optimizer into the Sesame NativeRDF SAIL. In evaluation Section 5.6, I provide details on the effectiveness of this optimization and its performance improvements.

Complexity

This optimization requires one full iteration over all keys, holds at most h nodes in main memory, never reads nodes from the optimal B-tree, and writes nodes in their optimal order to disk. Thus, this algorithms has a complexity of $O(N)$ regarding nodes read from the original and written to the optimized B-tree, as well as $O(h) = O(log_B N)$ regarding memory consumption.

Algorithm 3: Optimal B-tree Generation Strategy `optBTree`

Input: nodes sequence S, number of keys N, ordered keys sequence $K = (k_1, \ldots, k_N)$
Output: root node of optimal tree, new nodes sequence S

```
1  n ← new node ;                                  // create root node of this (sub-)tree
2  S ← S ∪ {n} ;                                   // and at it to the on-disk order sequence
3  h ← ⌈log_B(N + 1)⌉ − 1 ;                        // minimal B-tree height for N keys
4  if h = 0 then                                   // all keys fit into one node
5      for i ← 1 to N do                           // create this leaf node
6          append next key from K to node n ;      // append N keys
7      return n ;                                  // return created node as (sub-)tree
8  C ← ⌈(N − N_{h−1}) / (N_{h−1} + 1)⌉ + 1 ;        // minimal number of children of node n
9  (N_{c_1}, ..., N_{c_C}) ← optChildren(C, N − C + 1, h − 1) ;   // opt. children sizes
10 for i ← 1 to C − 1 do
11     c ← optBTree(S, N_{c_i}, it) ;              // create i-th child with N_{c_i} keys
12     append child c to node n ;                  // link from node n to child c
13     append next key from K to node n ;          // add next key to node n
14 c ← optBTree(S, N_{c_C}, it) ;                  // create C-th child
15 append child c to node n ;                      // link from node n to child c
16 return n ;                                      // return created (sub-)tree
```

Algorithm 4: Optimal Sub-Tree Set Generation Strategy `optChildren`

Input: number of sub-trees C, number of keys for all sub-trees N_C, height of sub-trees h
Output: optimal sequence of sub-tree sizes $(N_{c_1}, N_{c_1}, \ldots, N_{c_C})$

```
1  n_min ← C − 1 ;     // start with C-1 min full sub-trees hypothesis
2  n_full ← 0 ;                                    // and no full sub-trees
3  N_{c_r} ← N_C − (n_min · N_{min_h} + n_full · N_h) ;   // compute remaining keys
4  while N_{c_r} > N_h and n_min > 0 do
5      n_full ← n_full + 1 ;                       // one more full sub-tree is possible
6      n_min ← n_min − 1 ;                         // remove one min full sub-tree
7      N_{c_r} ← N_C − (n_min · N_{min_h} + n_full · N_h) ;   // update remaining keys
8  return (N_h, ..., N_h [n_full], N_{min_h}, ..., N_{min_h} [n_min], N_{c_r}) ;   // with N_{h−1} + 1 ≤ N_{c_r} ≤ N_h
```

5.5.5 A Better Index Selection Strategy

During the evaluation of a query plan, the index structures storing the statements are accessed with varying statement patterns based on the actual bindings known at evaluation time. As an example, let's reuse Join Tree 1 on page 78. Since no bindings are available for `SP1`, the statement pattern is resolved to (`? p ?`). Its evaluation on the statement index produces bindings for

?uri1 and ?type1. Then, SP2 and SP5 are both resolved to (s p ?).

For each such statement pattern, multiple B-tree statement indices can be used for evaluation. For a statement pattern (s p ?), a B-tree with the statement order spo and pso can be used. Considering statement patterns with context c, there are 16 possible statement patterns. The authors of [89] argue that six different statement orders suffice to cover all statement patterns. However, this analysis does not consider the dynamics of index employment caused by the sequence of evaluated statement patterns. The value of subject variable ?uri1 in the above example changes with each binding provided by SP1. With that, SP2 resolves to (s1 rdfs:label ?), (s2 rdfs:label ?), ... Now, different B-tree orders provide different evaluation performances for this dynamic. The pos index uses the property to identify the region of the index that contains statements with property rdfs:label. The subsequent evaluation of SP2 with varying bindings for s2 can benefit from caching, whereas the spo index looks up the subject for SP2 and fetches the region that contains all subjects *and* property rdfs:label, which is a much larger region than in the former case which significantly reduces cache hit probability.

I implemented a better index selection based on the variable binding order during join tree evaluation. Variables that are bound early are also used earlier in the B-tree order. As an example, when variable ?a is bound before ?b, ?a's values change with lower frequency in subsequent statement patterns than ?b. For a statement pattern (?a p ?b), the order pso is used, rather than pos. The advantage of the better index selection strategy is that the selected index will never perform worse than the static variant. However, it requires more than only six indices to cover all possible statement patterns.

Complexity

The better index selection strategy has constant complexity. Instead of only considering which fields of a statement pattern are bound, it also considers the timely order of the field's bindings.

5.6 Performance Evaluation of the LuceneSail

In the following, I report on a series of performance evaluations that measure the efficiency of the LuceneSail full-text search extension, as well as the impact of the general Sesame improvements.

A general problem in benchmarking hybrid queries performance is the lack of a publicly available full-text RDF data and query set. In literature, RDF full-text search is usually evaluated against individual data sets and queries, which are superficially described and never publicly available, which renders the comparison of results very difficult. For this reason, we published a full-text RDF benchmark described in Chapter 6. There, the LuceneSail is put into competition with other well-known and established RDF stores that provide full-text search capabilities. For now, I investigate particular pieces of our work and perform implementation-specific experiments. Firstly, I investigate the pure full-text search performance of the LuceneSail. Then, the effectiveness of our B-tree optimization is examined. Finally, I evaluate the complete system with a heavy load of complex hybrid queries to measure efficiency and performance improvements.

Full-text Search Performance A generally known and accepted benchmark allows for comparable and repeatable results [80, 103]. In the area of RDF indexing and querying, the Lehigh University Benchmark (LUBM) [86] is most commonly used. It can be used to generate arbitrarily large synthetic RDF data. However, it is not designed to be used in conjunction with complex full-text queries. The main reason for that is that the generated RDF literals are artificial one-term strings like "Course26" or "Publication3Common". By using such textual data, keyword search degrades to simple complete string matching.

We basically want to estimate the typical performance of the full-text search capabilities of LuceneSail. For this, we take a large RDF graph that primarily contains literals: the Wordnet RDF graph[17]. This graph represents the relations between English nouns, verbs, adjectives, and adverbs, and is organized into sets of synonyms, each representing one fundamental lexical concept. Nodes usually have multiple literals containing one or only a few terms like "cognition" or "knowledge" and one longer multi-term literal like "the psychological result of perception and learning and reasoning". Further, this graph contains a large amount of structural information. Altogether, the Wordnet RDF graph contains over 473,000 statements, including over 273,000 statements with literals. The corresponding NTriples file is 71 MByte in size. After indexing the RDF graph, the Sesame NativeRDF store occupies 52 MByte, while the corresponding Lucene index is 47 MByte in size. For our experiments we index and store all predicates.

We then issue 100,000 queries, each containing one, two or three random terms that occur in the indexed RDF literals. Each query is issued only once to avoid caching effects. As the query response time we consider the time between issuing the query and fetching all results. We evaluate pure full-text queries and retrieve the score and URI of the matching resources. In Figure 5.7 you can see that with increasing result set size the average response time increases linearly, mainly caused by the linear complexity of fetching the results from disc. However, the response time is always below 50 ms for less then 1,000 results. The number of terms does not have a significant influence on the overall response time, so the performed joins must be negligible in time. The area between the σ- and the 2σ-lines, respectively, illustrate the deviation of the response time.

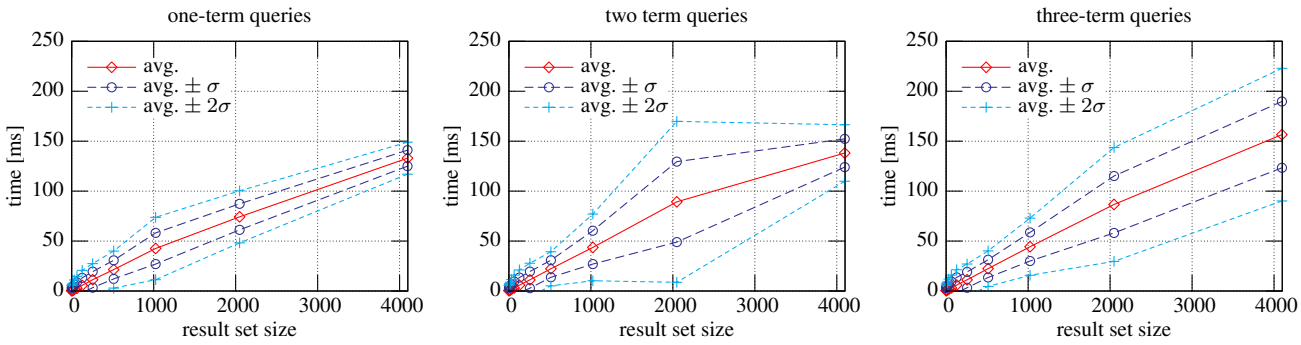


Figure 5.7: Pure full-text search performance of LuceneSail.

[17]Wordnet RDF: `http://www.w3.org/2006/03/wn/wn20/`.

B-tree optimization As the next evaluation, we examine the effectiveness of the B-tree optimization and its impact on iteration operations. For this, we index a significant portion of the DBpedia data set version 3.5.1 comprising ≈ 89 million statements and 226 MByte serialized in RDF/XML. Figure 5.8 and Table 5.1 depict some statistics of the original and optimized B-trees indexing these statements in the respective statement order. All original B-trees are created by adding statements in the order given by the RDF/XML serialization. Due to the statement orders of the trees, the same statement is inserted at different location in the trees. This leads to individual tree reorganization operations, which causes the variation in file size, number of nodes and node position within the tree file. All optimized trees have the same structure (number of nodes and number of keys in individual nodes), because it only depends on N, which is identical for all six B-trees.

statement order	size in MByte		number of nodes		statements per node		full iteration in seconds	
opsc	3.01	1.75	1,577,963	917,422	56.40	97.0	12,690.71	98.98
ospc	2.95		1,545,074		57.60		6,505.49	75.51
posc	3.09		1,621,781		54.87		4,793.23	106.46
psoc	3.45		1,809,510		49.18		7,963.20	81.70
sopc	3.44		1,801,505		49.40		161.03	112.20
spoc	2.57		1,349,718		65.93		672.92	83.10

Table 5.1: Statistics on six original (left value of each column) and optimized (right) B-trees after indexing ≈ 89 million statements from the DBpedia data set.

With one full iteration over the original and optimized B-tree, we measure the impact of the optimization on iterations. During an iteration over trees, nodes are fetched from disk in depth-first sequence. The node's position within the B-tree file, the timestamp when a node is fetched, together with the node's position in the iteration sequence are recorded. Figure 5.9(a)–5.9(c) plot those data for three statement orders. The x-axis depicts the index of a node in the fetch sequence, whereas the y-axis refers to the node's position in the B-tree file. Long vertical lines refer to large I/O seek distances[18], which take more time than fetching a node that is located closely to the previous one. The time that is consumed by a far seek causes a steeper slope of the time curve, because more time is needed to fetch a node.

In contrast, the optimized B-tree depicted in Figure 5.9(d) has minimal iteration performance and I/O seek distances. The time for a full iteration is also given in Table 5.1. The variation of full iteration time can be explained by competing disk operations during the evaluation and file fragmentation caused by the file system. We can observe that the measured time is significantly better than for original B-trees, once even two orders of magnitudes. All optimal trees yield similar iteration performances, where the original trees span almost two orders of magnitude of variation.

[18] I assume the B-tree file to be stored into subsequent blocks on the hard disk (*i. e.*, not fragmented). Otherwise, this would break the correlation between node positions within the B-tree file and on-disk I/O seek distances.

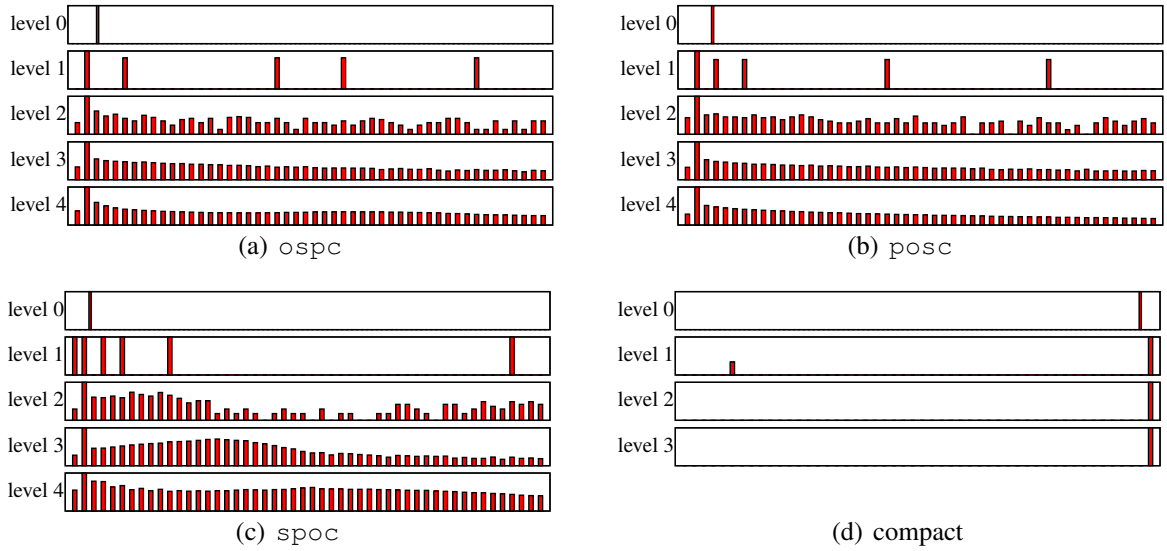


Figure 5.8: Node size histograms (equiv. to *detailed profiles* [154]) of three original and one compact B-trees showing the probability of a node (the root) to have 48–97 (1–97) statements. The y-axis is logarithmic, probabilities of one level sum up to 1. Note that the compacted trees of different B-trees of the same size are structurally identical.

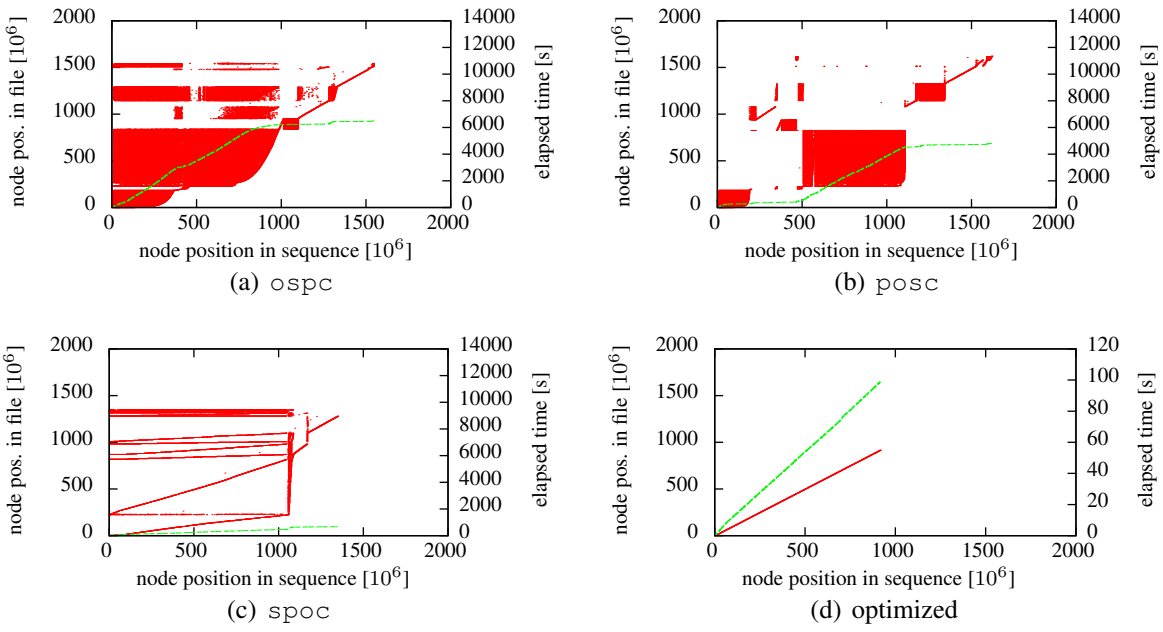
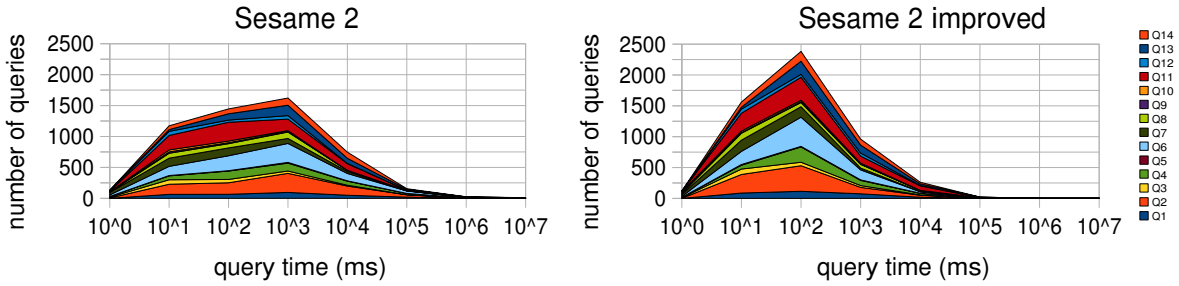


Figure 5.9: A full iteration over three original and one optimized B-trees. A red point denotes an individual node, the green line plots the time that passes by during the iteration. Note the different time scale on the y-axis for the optimized index.

Hybrid Query Evaluation The last evaluation targets at the performance improvement of our general Sesame enhancements. To this end, complex hybrid queries are generated using the QUICK system [192]. For a keyword query given by the user it finds all types and properties of a given ontology that match single keywords. In a second phase, all possible hybrid queries of a particular size are generated according to the ontology. This experiment uses the RDF data set of the Internet Movie Database[19] (IMDb). Its ontology contains 5 types and 10 properties. The data set provides more than 10 million instances and 40 million facts.

To estimate the performance in real-world settings, real user queries are extracted from the query log of the AOL search engine. Here, approximately 3,000 user keyword queries that brought the user to the IMDb website can be extracted. Unfortunately, most of these queries are rather simple: they only contain a single type. Therefore, a manual investigation is necessary to select those referring to more than two concepts. This yields 100 user queries.

From these, 14 queries are randomly selected and used by the QUICK system to generate the semantic query space with a maximum size of 7 links. This generates 5,305 hybrid queries each having more than 100 results. All these queries are evaluated against the original and the improved Sesame RDF store with LuceneSail, to measure the general performance improvement. The experiments were conducted on a 3.60 GHz Intel® Xeon® server. Throughout the evaluation, less than 1 GB memory was occupied.



(a) original Sesame (b) improved Sesame

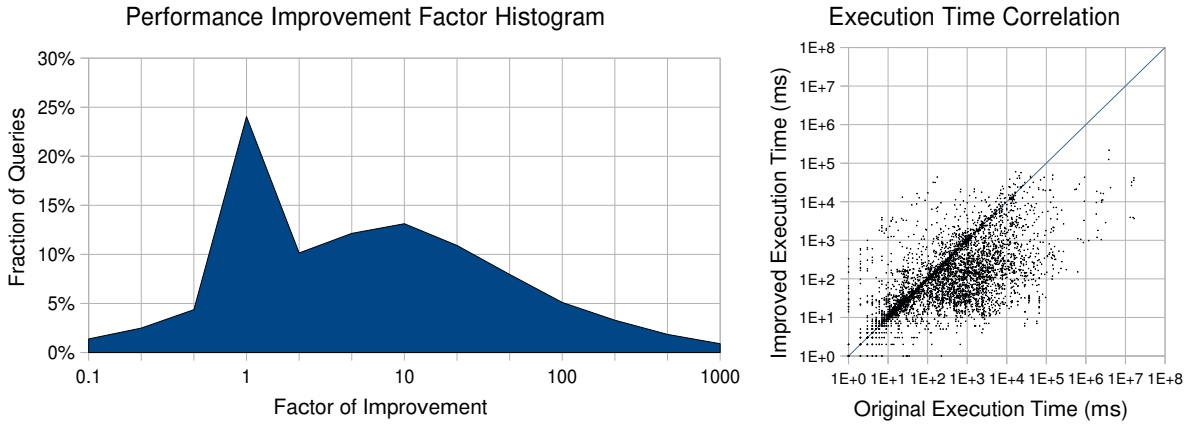
Figure 5.10: Histogram of original and improved Sesame query evaluation times

The query time of these queries is depicted as a histogram in Figure 5.10(a) for the original Sesame, and in Figure 5.10(b) for our improved version. The different colors refer to the particular user query the queries are generated from. One can see that the average evaluation time of the queries improve by one order of magnitude, from 22.7 s down to 1.03 s.

This improvement can also be seen in the histogram of improvement factors shown in Figure 5.11(a). For each query, an improvement factor is computed by diving the original execution time by the improved one. All queries except those taking less than 100 ms in both cases constitute the histogram. Below 100 ms, the measured execution time is very inaccurate due to the low time span. The factors of those queries would add significant noise to our subsequent analysis and are therefore filtered out.

[19]The Internet Movie Database (IMDb): `http://www.imdb.com/`.

The high peak at the factor of 1.0 refers to all queries that do not significantly change in their execution time. Most queries improved by one order of magnitude, whereas a significant proportion of queries even improved by factor 100 or even 1,000. However, also a significant number of queries experience a degraded execution time. These queries suffer from the wrong query plan generation, due to inaccurate cardinality estimations.



(a) Improvement Factor Histogram

(b) Execution Time Correlation

Figure 5.11: Histogram of performance improvement factors and the correlation between the original and improved execution time.

To better assess the significance of these low improvement factors, we plot the original execution time against the improved one in Figure 5.11(b). One point refers to one query. On the left of the $y = x$ line, all queries are located that experience a worse execution time in our improved system. One can see that most of these queries originally take less than one second, where a degradation by factor 10 still yields to an agreeable performance. In contrast, more improved queries originally take more than ten seconds than those being degraded.

5.7 Conclusion and Outlook

I have presented our LuceneSail, a combination of structured queries (DB) and full-text queries (IR). The essence of this approach is to embed queries for the full-text of RDF literals into structured queries using virtual properties. The results of these queries are computed by combining an RDF store (Sesame) and a highly efficient full-text search engine (Apache Lucene). The score and text snippet results from the IR query are integrated with results from the structured query, whereas in related work this information is often omitted. Several optimizations are done to the hosting RDF store Sesame, to better integrate the evaluation of full-text query parts and improve general structured query evaluation. With our performance evaluation based on general-purpose data sets (Wordnet, IMDb, and DBpedia RDF graphs), we demonstrate fast response times of our system, as well as significant performance improvements.

This work's general approach of combining structured with full-text search can be replicated by others and adopted to other RDF storage systems. We envision that future Desktop and Web Search Engines will be based on a combination of SPARQL and full-text retrieval, for which this work is a needed input.

During implementation and evaluation of our full-text extension of Sesame, we identified some potential performance improvements. The primary improvement can be achieved by a joint URI handling scheme. Hybrid queries are evaluated against two indices (Sesame and Lucene). In order to join results, the URI of matching resources have to be compared. Unfortunately, the URI needs to be read from the actual Lucene documents that match the full-text query part, which requires I/O operations. The same holds for the URIs of the Sesame store. However, since not all of these fetched URIs will match both query parts, irrelevant and unnecessary I/O operations are performed. The Lucene document IDs are in contrast available without additional effort, so the join could be performed on them. A mapping between Sesame's internal URIs representation and these Lucene document IDs would yield a better join performance. However, such a mapping has to be updated after each Lucene index optimization.

The full-text extension of Sesame is available as free software and was used in the research projects *Beagle*$^{++}$ [122], Gnowsis [165], and NEPOMUK [131], several research activities [46, 64, 168], as well as in the industrial domain by Aduna Autofocus and ABC News[20].

[20] ABC News: `http://abcnews.go.com/`.

Chapter 6

Benchmarking Hybrid Queries

Developers and researchers evaluate the performance of their systems with the help of benchmarks. To obtain comparable results, such a benchmark usually comprises a well defined data set and work load that mimics realistic usage patterns [80, Ch. 1] [103]. For evaluation of RDF stores, a number of benchmarks have been proposed which measure the performance of structured queries [30, 86, 169]. None of them, however, addresses full-text search capabilities. I consider this to be the primary reason for RDF store developers to test their full-text search implementations with their own ad hoc benchmarks [91, 124, 195]. This circumstance renders efficiency evaluations difficult to be repeated or compared. To the best of my knowledge, a commonly available RDF full-text benchmark was missing prior to this work.

In this chapter, I propose a benchmark that measures performance and feature richness of full-text search facilities of RDF stores. For this, I extend the well-known *Lehigh University Benchmark* (LUBM) [86] with full-text content and hybrid queries. With the help of this new benchmark, I evaluate the established and freely available RDF stores and discuss the results.

In particular, this work makes the following contributions:

1. I identify properties of data sets needed for an RDF full-text benchmark, taking into account real full-text characteristics such as its term distribution.

2. I extend the LUBM benchmark data set, propose an algorithm for scalable generation of synthetic full-text content and make the implementation publicly available[1].

3. I design RDF full-text queries to investigate different aspects of full-text search on RDF, and evaluate well-known open source RDF stores with this benchmark (in alphabetical order): *Jena*, *Sesame2*, *Virtuoso*, and *YARS*.

The results show insights of RDF stores regarding basic full-text queries (classical IR queries) as well as hybrid queries (structured and full-text queries). These results are valuable for selecting the right RDF store for specific applications. They further reveal the need for performance improvements for certain kinds of queries.

This work is structured as follows. After discussing the related work of benchmarking in general and RDF stores in particular in Section 6.1, I define objectives and desired features of an RDF full-text benchmark, and describe my extension of the LUBM benchmark in Section 6.2. Section 6.3 presents the evaluation of popular RDF stores against the proposed RDF full-text benchmark. Finally, I close this chapter with my conclusions in Section 6.4.

[1] LUBM_{ft}: `http://www.l3s.de/~minack/rdf-fulltext-benchmark/`.

6.1 Related Work

In the software engineering domain, benchmarks are used to assess the relative performance and absolute feature-richness of a target system [80]. For this, well designed standard tasks mimic a particular type of workload [103, Ch. 4]. The *Full-Text Document Retrieval Benchmark* [80, Ch. 8] (FTDR) generates workload as a mixture of searching, retrieving documents from the system, and adding documents. In contrast, our benchmark only investigates search.

In the area of relational databases, the *TEXTURE* benchmark [73] investigates performance of relational text processing workload. It randomly generates queries based on certain term occurrence characteristics. In contrast, our work provides fixed queries that can repeatedly be evaluated and studied for different RDF stores and hardware configurations. Similarly to our benchmark, TEXTURE synthetically generates full-text but only maintain one global word distribution.

In the area of RDF stores, a number of benchmarks are available. The *Berlin SPARQL Benchmark* (BSBM) [30] generates full-text content and person names. The SP^{2}Bench [169] uses a data set that refers to the structure of the DBLP Computer Science Bibliography[2]. Both benchmarks pick terms from dictionaries with uniform distribution, whereas our benchmark employs a realistic long-tail distribution of terms. Further, the query sets of these benchmarks do not contain full-text or hybrid queries. Enriching these benchmarks with real world full-text content and full-text queries in the way our work extends the LUBM benchmark would indeed be possible. To the best of our knowledge, this full-text extension of the LUBM benchmark is the first full-text Benchmark for Semantic Web systems.

Due to the lack of RDF full-text benchmarks, in [124] we employ the RDF version of Wordnet[3] and query for randomly selected terms. However, this data set does not easily scale up to an arbitrary size. Similarly to our approach, the authors of [91] generate a data set, where literals are generated by keywords randomly selected from a dictionary. Unfortunately, the authors neither sufficiently report on this benchmark in order to reproduce results or reuse the benchmark, nor do they apply that benchmark on other RDF stores.

6.2 A Full-text Extension for LUBM

This RDF full-text benchmark is based on the *Lehigh University Benchmark* (LUBM), which I introduce in the following section. Its value for benchmarking search over RDF data is discussed and missing features regarding full-text search are outlined. Finally, the requirements on data set generation, as well as new full-text queries are derived for our RDF full-text benchmark.

[2]DBLP Computer Science Bibliography: `http://dblp.uni-trier.de/`
[3]Wordnet RDF: `http://www.semanticweb.org/library/`

6.2.1 LUBM Overview and Discussion

The *Lehigh University Benchmark*[4] (LUBM) [86] is a very frequently used synthetic benchmark for RDF stores [89, 91, 195]. It provides the *Univ-Bench Artificial data generator* (UBA) to generate data sets of arbitrary size, and a set of 14 queries. Its data set complies with the *Univ-Bench ontology*[5] describing universities. It contains 43 classes, such as `ub:University`, `ub:Department`, `ub:Professor`, and `ub:Publication`. It further contains 25 object properties (pointing to resources), as well as 7 data type properties (pointing to literals), such as `ub:name` and `ub:publicationAuthor`.

Data sets are generated using a scaling factor N which refers to the number of universities being synthesized and therefore determines the size of the generated data set. In addition, a seed M for the random number generator can be specified to allow repetitive regeneration of the same synthetic data set. Generated data sets are referred to as LUBM(N, M). The notation LUBM(N) indicates that M is set to 0. Similarly, I name this full-text benchmark LUBM$_{\text{ft}}$ and refer to the full-text extended data set of size N as LUBM$_{\text{ft}}(N)$.

Even though the LUBM benchmark was primarily designed to evaluate OWL and DAML+OIL capabilities of RDF stores (see Section 2.1), it was also very often used to benchmark RDF stores and RDF related systems without OWL or DAML+OIL support [89, 91, 195]. This may be due to the very easy generation of arbitrarily sized RDF data sets and the familiar ontology domain. As LUBM, this benchmark works perfectly on stores without any kind of reasoning.

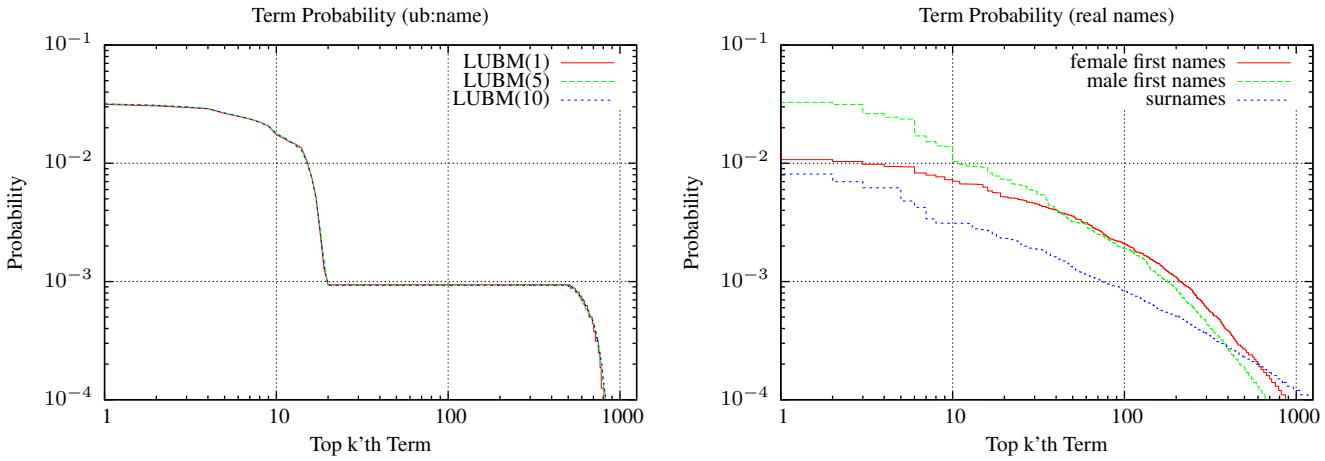


Figure 6.1: (Left) Term distribution of the `ub:name` predicate. (Right) Term distribution of first names and surnames provided by the 1990 U.S. Census.

[4]The Lehigh University Benchmark (LUBM): `http://swat.cse.lehigh.edu/projects/lubm/`.

[5]Throughout this paper I refer to the Univ-Bench namespace using `ub`, which resolves to `http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl`.

During the generation process, instances and relations are created on the basis of uniform distributions. For instance, a university contains 15 to 25 departments, each employing 7 to 10 full professors, whereas each authored 15 to 20 publications. This equips the LUBM data sets with quite a complex structure with well defined connectivity properties. However, the literal values lack such sophisticated characteristics. The name of a person, for instance, is a one term literal like `"FullProfessor0"`. Full-text search for this term provides exactly one match for each university department. Finally, the `ub:name` property is used for all kinds of resources.

This results in a mixture of uniform distributions of this property's terms which is highly unrealistic. Figure 6.1 shows the distribution of `ub:name`'s terms (left), and a real world distribution of first names and surnames (right). One can see that surnames and male first names are even power law distributions, as many real-world distributions are [139, Fig. 4].

6.2.2 Full-text Content Generation

The proposed full-text extension modifies aspects of the LUBM data set twofold. Firstly, names are generated in such a way that they follow realistic term distributions. Secondly, to measure performance on large text literals, full-text content is added to publication instances. To implement this extension, two sources of literals were added to the LUBM data generator, 1) the *Name Generator*: adding real person names and 2) the *Document Generator*: providing publication content. The enhanced data sets contain the same statements as the original LUBM data sets, extended by the `ub:firstname`, `ub:surname`, `ub:fullname`, as well as the `ub:publicationText` predicate, providing the respective content.

Name Generatator

The first new source of literals is the *Name Generator*, which produces random names consisting of a first name and a surname. The generated terms follow a real distribution of names. As input for the term frequencies we use the data provided by the U.S. Census Bureau from the 1990 census[6]. This data set provides probabilities for the top $\approx$ 1,200 male and $\approx$ 4,300 female first names. It also contains the top $\approx$ 89,000 surnames, whereas only the top $\approx$ 19,000 surnames have sufficient frequency information.

Document Generator

The second source of literals with realistic term distributions is the *Document Generator*. Since LUBM is a synthetic benchmark that produces data of any specified size, the content generation also needs to scale. Therefore, we cannot directly use a real-world document collection as content. Instead, we use a Probabilistic Topic Model [179] to learn features that characterize such a collection. We then generate a synthetic set of documents which exhibits the learned real-world characteristics. This approach preserves the desired scalability feature of LUBM and at the same time ensures realistic term frequency distributions of the generated collection.

[6]Frequently Occurring First Names and Surnames From the 1990 Census: `http://www.census.gov/genealogy/names/`.

To learn a realistic topic distribution of documents, we apply *Latent Dirichlet Allocation* (LDA) [179]. With LDA, a document is modeled as a mixture of topics, and a topic is modeled as a distribution over words. We reuse the Matlab® implementation of the LDA model provided by Steyvers and Griffiths [179]. As a training set we employ the NIPS document collection[7], consisting of 1,740 papers written by 2,037 authors with a total of 2,301,375 term occurrences of 13,649 unique terms. From the topic distribution of documents, a topic co-occurrence probability is derived: a topic with a high probability regarding a document gains a higher co-occurrence probability with all other topics of that document than a topic with a low probability would gain. This allows us to respect the phenomenon that terms tend to co-occur with some terms more likely than with others [52], whereas drawing terms from a single term distribution would not consider this phenomenon.

Given the topics and their co-occurrence, we first assign a dedicated topic to each department. Then, we distribute topics co-occurring with the department topic among the faculty staff of the department, according to their probabilities. Together with the department topic, the topics of the authors determine the topic mixture of a publication. This topic assignment clusters publications around departments by means of used terms: each department and author has its own specific vocabulary. This better reflects the phenomenon of different vocabularies among universities, departments, and authors. Table 6.1 gives an overview of the resulting LUBM_{ft} data sets.

	$\text{LUBM}_{\text{ft}}(1)$	$\text{LUBM}_{\text{ft}}(5)$	$\text{LUBM}_{\text{ft}}(10)$	$\text{LUBM}_{\text{ft}}(50)$	$\text{LUBM}_{\text{ft}}(100)$
Universities	1	5	10	50	100
Person Names	8,330	51,955	106,409	555,815	1,120,834
Publications	5,999	37,854	76,529	402,142	808,741
Unique Terms	17,611	28,171	32,438	36,456	36,518
Terms	6,032,320	38,061,820	76,954,636	404,365,260	813,224,336
Statements	134 k	840 k	1.7 M	8.96 M	18 M
Size (XML)	56 MB	356 MB	719 MB	3.8 GB	7.6 GB

Table 6.1: Statistics of the LUBM_{ft} data sets.

6.2.3 Full-text Test Queries

The generated full-text data set needs to be complemented with related queries to form a complete benchmark. The goal is to design these queries in a way that evaluating them against the LUBM_{ft} data sets provides insights into the strengths and weaknesses of RDF stores regarding full-text search. We deem query plan optimization an integral part of an efficient query evaluation. Therefore, we follow the same principle as LUBM in which query patterns are stated in descending order of their cardinality. This gives RDF stores the maximum possible opportunity to optimize the query plan, and reduces the probability that a store by chance executes a better query plan than another store.

[7] Author-Topic Model: NIPS: `http://www.datalab.uci.edu/author-topic/NIPs.htm`.

Our benchmark queries are subdivided into three sets, each targeting a different area:

Basic IR performance: These queries are equivalent to classical IR queries, *i. e.*, keyword queries without any semantic relation.

Semantic IR performance: This set of queries targets at the performance of the combination of keyword queries and semantic relations — hybrid queries. Furthermore, this investigates the quality of full-text search integration into the SPARQL evaluation process.

Advanced IR features: Finally, this set of queries explores advanced IR features like boolean, phrase, or wildcard queries, score or snippet retrieval.

Queries In the following, we present the three query sets, in which the queries get more and more complex, and where subsequent queries build on the results of previous ones. Note that some queries have variants, *e. g.*, Query 1 is evaluated with different keywords to investigate the impact of those different keywords for the same data set size. The two variants are denoted as Query 1.1 and 1.2.

All queries focus on full-text performance, their *combination* with structured queries and their *integration* into the query evaluation process. We do not investigate the performance of the structural part of the queries. The performance of structured query parts is sufficiently investigated by RDF benchmarks discussed in Section 6.1.

Basic IR Performance

The first set of queries that we design will only contain queries that can equivalently be expressed in pure IR queries. Furthermore, only those IR features are exploited that are expected to be the minimum set of features all RDF stores with full-text search support, namely keyword and phrase queries.

Q1:	"All resources matching the keyword k_i". $k_i \in \{$'engineer', 'network'$\}$
Q2:	"All resources matching k_i in `ub:publicationText`". $k_i \in \{$'engineer', 'network'$\}$
Q3:	"All resources matching 'network' or 'engineer' in `ub:publicationText`".
Q4:	"All resources matching the phrase 'network engineer' in `ub:publicationText`".
Q5:	"All resources matching keyword 'smith' / having literal "Smith" in `ub:surname`".

Table 6.2: Basic IR Queries.

Query 1 starts with a simple one-keyword query. Its performance is of interest, since in the more complex benchmark queries the evaluation of keyword query conditions produces equivalent intermediate result sets with a proportional performance and scalability impact. One infrequent term ('engineer') and one very frequent term ('network') is used to investigate the influence on the result set cardinality. For $\text{LUBM}_{\text{ft}}(1)$, the result set cardinality for these terms is 40 and 1,013, respectively.

In Query 2, the terms are restricted to match in the `ub:publicationText` predicate. The result set is the same, so the difference in performance compared to Query 1 is caused by this additional constraint only. This query is equivalent to fielded keyword search in IR. Result sets of two keywords have to be looked up and merged to evaluate Query 3, which is the union of both result sets. To evaluate a phrase query of the two keywords, the position information of each occurrence of both keywords has to be processed by the full-text search engine for Query 4. Query 5 finally allows to compare keyword look-up performances with literal look-up, since both queries are semantically equivalent and have the same result set.

Semantic IR Performance

The following queries contain structural conditions to investigate their impact on the query evaluation performance. These queries show the quality of integration of IR features with general SPARQL query processing.

Q6:	"All `ub:Publications` matching 'engineer' in `ub:publicationText`".
Q7:	"All `ub:Publications` and their `ub:title` matching 'engineer' in predicate `ub:publicationText`".
Q8:	"All `ub:Publications`, their `ub:titles`, and the `ub:FullProfessor` author's `ub:fullname`, matching 'engineer' in `ub:publicationText`".
Q9:	"All resources matching 'engineer' in `ub:publicationText` and all resources matching 'smith' in `ub:fullname`, being connected via the predicate `ub:publicationAuthor`".
Q10:	"All resources matching 'network' in `ub:publicationText` and all resources matching 'engineer' in `ub:publicationText`, that are both connected via `ub:publicationAuthor` to the same `ub:FullProfessor`".
Q11:	"All distinct `ub:FullProfessors` matching 'smith' in `ub:fullname` that authored both, resources matching 'network' and resources matching 'engineer' in the `ub:publicationText` property".

Table 6.3: Semantic IR Queries.

From Query 6 to Query 8, we increase the number of structural constraints of the query. The result set of the keyword condition has to be joined with more and more statement patterns. This allows us to evaluate the impact of structural query parts. The next step is the extension to two-keyword queries, where each keyword matches a different resource in the query graph. In Query 9, both matching resources are tested whether they are connected via a `ub:publicationAuthor` predicate. Query 10 interconnects both matching resources via a `ub:FullProfessor` resource. Finally, this query is extended to three keywords, where the intermediate full professor additionally has to match the third keyword (Query 11). This requires three result sets of the full-text search to be joined via structural restrictions.

Advanced IR Features

The final query set investigates the existence of certain advanced IR features. In contrast to the features evaluated by Queries 1 to 5, the advanced features are not expected to be supported by all RDF stores offering full-text search.

Q12:	"All resources matching 'network' **and** 'engineer' in `ub:publicationText`".
Q13:	"All resources matching 'network', **but not** 'engineer' in `ub:publicationText`".
Q14:	"All resources matching 'network' and 'engineer' in `ub:publicationText`, both keywords appearing within a distance of at most 10 words of each other".
Q15:	"All resources matching wildcard keyword 'engineer*' in `ub:publicationText`".
Q16:	"All resources matching wildcard keyword 'engineer?' in `ub:publicationText`".
Q17:	"All resources matching fuzzy keyword 'engineer~0.8' in `ub:publicationText`".
Q18:	"All resources and their relevance to the keyword 'engineer' in predicate `ub:publicationText`".
Q19:	"All resources and a snippet matching keyword 'engineer' in predicate `ub:publicationText`".
Q20:	"The top-10 resources with their relevance to the keyword 'network' in `ub:publicationText`".
Q21:	"The resources and their relevance to 'network' in predicate `ub:publicationText`, that have a relevance of > 0.75".

Table 6.4: Advanced IR Queries.

There are tests for conjunction (Query 12) and negation (Query 13) of full-text related conditions, for a proximity query (Query 14), where both keywords have to appear in a distance of at most 10 words, and for wildcard queries (Queries 15 and 16).

Query 17 matches all terms that have a similarity of 0.8 to the given keyword. The similarity measure is application dependent, so for different implementations, different similarity values may be used to retrieve a comparable result set size. The relevance score and a snippet is retrieved by Queries 18 and 19, respectively. Some RDF stores allow to limit the results of a full-text search to top-k, or to all results that exceed a certain relevance score. These features are tested by Queries 20 and 21. If these features are not inherently supported, they can be mimicked by a SPARQL LIMIT or FILTER operation. In that case, a similar performance to Query 2.2 is expected.

6.3 Evaluation

After having generated the enhanced LUBM_{ft} data set and designed 21 queries, a number of well-known open source RDF stores are evaluated using this RDF full-text benchmark. The following RDF stores (in alphabetical order) are chosen: Jena [37], Sesame2 [34], Virtuoso[8],

[8]Virtuoso Open-Source Edition: `http://virtuoso.openlinksw.com/wiki/main/Main/`

and YARS [89]. Since Jena provides a large number of different backend stores, preliminary tests identify Jena with TDB to be the fastest configuration.

In detail, the following configurations are used, which will further be referred to using these abbreviations:

Jena: Jena 2.5.6, ARQ 2.5.0, Lucene 2.3.1, TDB 0.6.0

Sesame2: Sesame 2.2.1, NativeStore, LuceneSail (Hits-Set) 1.3.0, Lucene 2.3.2

Virtuoso: Virtuoso 5.0.9

YARS: YARS post beta 3, Lucene 1.9.1

All Java-based RDF stores employ the Lucene[9] full-featured text search engine Java library, the state-of-the-art Java implementation for Information Retrieval.

6.3.1 Methodology

The evaluation is conducted over $\text{LUBM}_{\text{ft}}(N)$ with $N \in \{1, 5, 10, 50\}$. The *LUBM benchmark test tool* (UBT) version 1.1 is reused to perform the tests. However, its program sequence is modified towards the following behavior.

The queries are designed in an incremental manner with subsequent queries building on findings of previous queries. This means that the result set of an early query is sometimes the intermediate result set of a later query. Due to RDF store specific caching mechanisms and file-system cache managed by the operating system, subsequent executions of the same query, as well as similar queries, benefit from those "warmed up" caches. In order to suppress these side-effects, the RDF store caches are cleaned by restarting the store for each query, and clearing the file-system cache. Then each query is evaluated six times, where the first duration is considered as "un-cached" and the subsequent durations are considered as "cached". This provides insights in the performance of each store in case of insufficient and sufficient memory for caching. Queries that exceed a limit of 1,000 s are excluded from evaluation of larger N. The duration of a query is defined as the time that passes from parsing the query until no more results are provided by the RDF store.

For each RDF store and LUBM_{ft} data set, all 21 queries are evaluated in this manner. This whole process is repeated 5 times. In the end there are 5 evaluation times for each query referring to the "un-cached" case, as well as 25 times under the presence of "warmed up" caches.

For the evaluation I used a GNU/Linux machine with a 2 GHz AMD Athlon™ 64 bit Dual Core Processor, 3 GByte RAM, and a RAID 5 array. The UBT Tester ran with Java™ SE Runtime Environment 1.6.0_10 and 2 GB Memory.

[9] Apache Lucene: `http://lucene.apache.org/`

6.3.2 Results

In the following, I present the results of this benchmark evaluation. For easier comparison, all figures share the same layout. Each figure depicts one benchmark query and aggregates all RDF stores. Along the x-axis the different data sets are aligned by increasing scaling factor N. The logarithmic y-axis depicts the evaluation time of a query in *milliseconds*. The bars of the RDF stores are subdivided into two parts: the whole bar represents the evaluation time with cleared caches, whereas the lower part of the bar illustrates the evaluation performance under the existence of "warmed up" caches.

Basic IR Performance

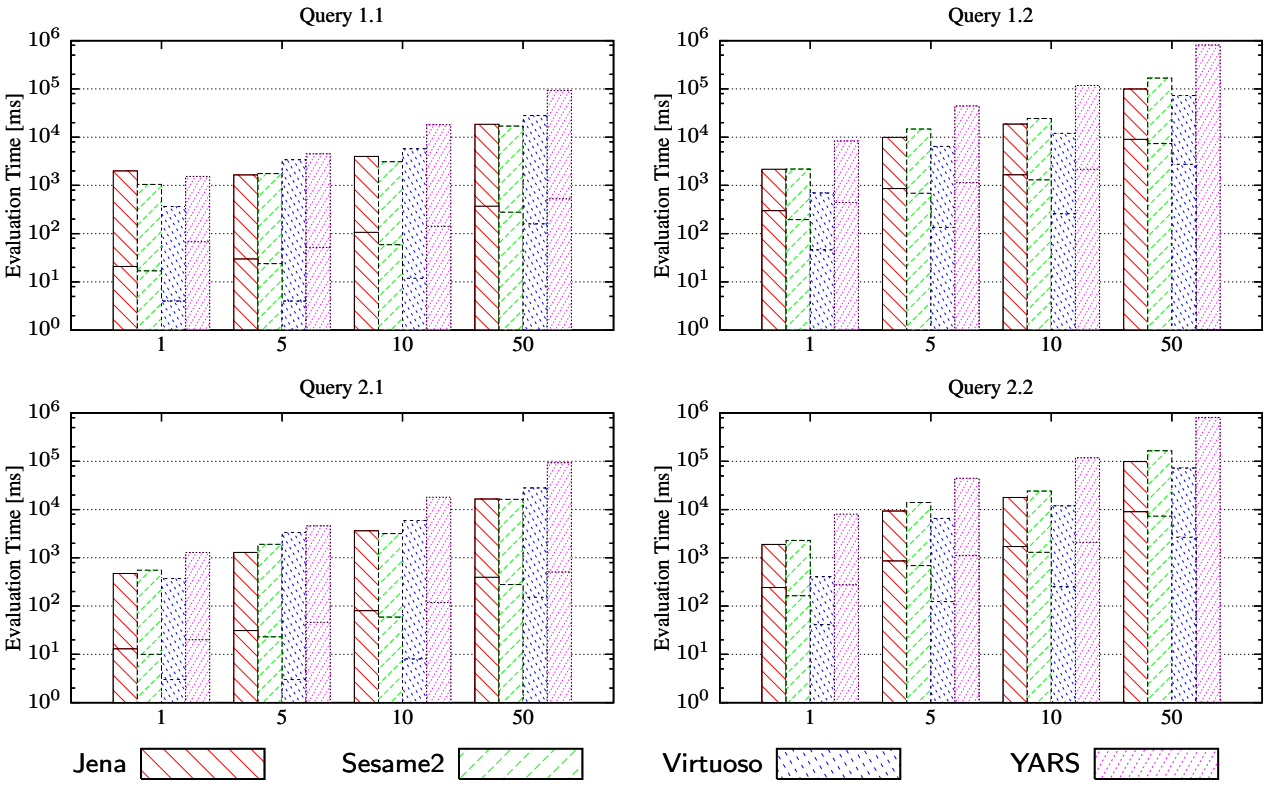


Figure 6.2: Single keyword queries without and with predicate binding.

Figure 6.2 depicts the one-keyword Queries 1.1 and 1.2, which represent the basis of full-text search. For Query 1.1, Jena is slower than Sesame2, but Jena can better deal with the larger result set of Query 1.2, and can therefore exhibit a small performance advantage. Virtuoso, which is very fast for the smallest data set, performs worse than Jena and Sesame2 with increasing data set size for small result sets (Query 1.1), but performs better for larger result sets (Query 1.2). For both queries, YARS reveals the worst performance. For all queries, Virtuoso has an unrivaled performance with "warmed up" caches.

The results of Queries 2.1 and 2.2 exhibit for all stores the same performance as for the Queries 1.1 and 1.2. Therefore, the restriction imposed by Query 2 does not cause any computational challenge to any of the stores.

With Query 3 the result set size does not significantly increase compared to Queries 1.2 and 2.2, but multiple keywords are used in a boolean *OR* semantic. As we can see in Figure 6.3, for Query 3 as for Query 1.2, all three stores that use Lucene exhibit the same scalability, whereas Virtuoso does not support this query. Compared to Sesame2, Jena also yields a better performance for Query 4. Here, Virtuoso is even slower than Sesame, but it shows the lowest response times in case of "warmed up" caches. YARS does not allow for phrase query articulation. Looking at Queries 5.1 and 5.2, all RDF stores have a better performance evaluating the latter one.

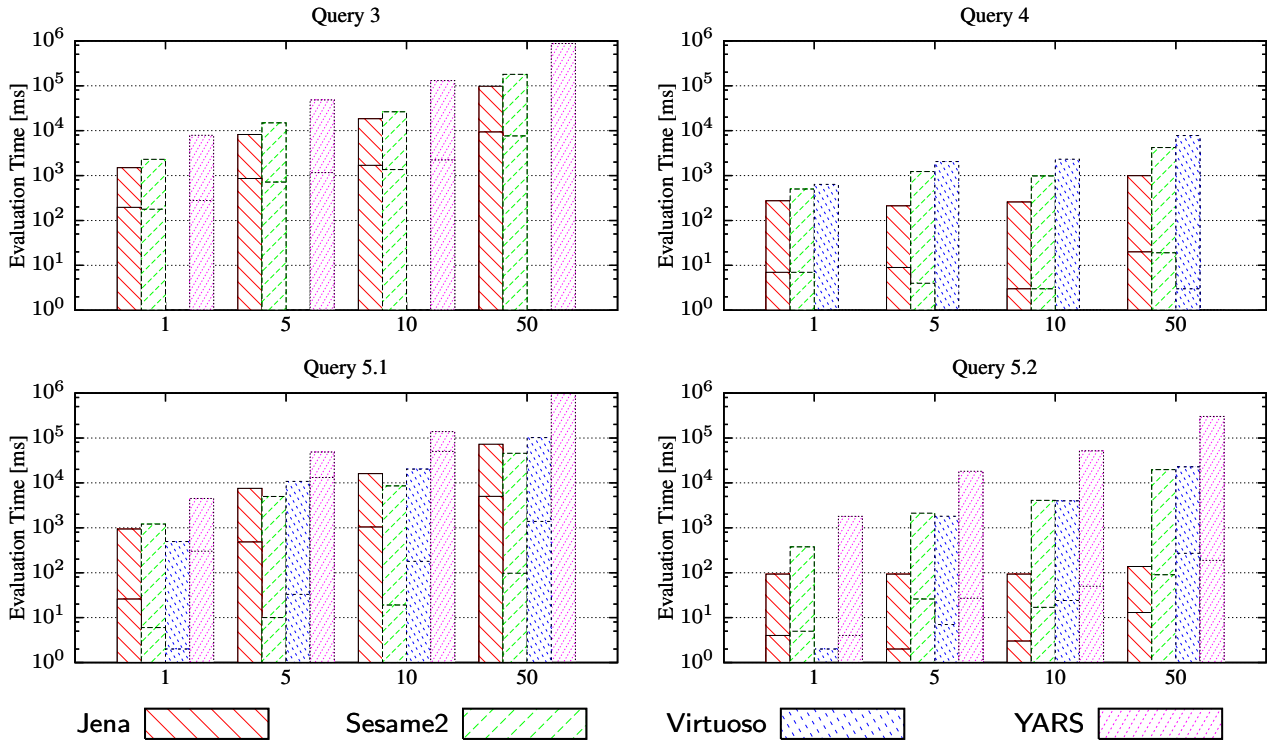


Figure 6.3: Multiple-keyword, phrase, and keyword vs. literal queries.

Discussion From Queries 1 and 2 we can see that none of the stores is challenged by missing or given predicate bindings for the full-text search. All stores are robust against both situations.

Regarding basic IR queries, Jena is slightly faster than Sesame2. Both use Lucene as their full-text search engine, so the overhead of the integration of the IR engine into the RDF evaluation is obviously slightly higher in Sesame2. YARS also uses Lucene, but is much slower. This may be due to the outdated lucene version they use. Virtuoso is for all but one case the slowest RDF store. Further, it only supports a subset of the basic IR features. Queries 5.1 and 5.2 reveal that a literal look-up is faster in all stores than an equivalent term look-up in the full-text index. This indicates potential improvements of the systems regarding their full-text integration.

Semantic IR Performance

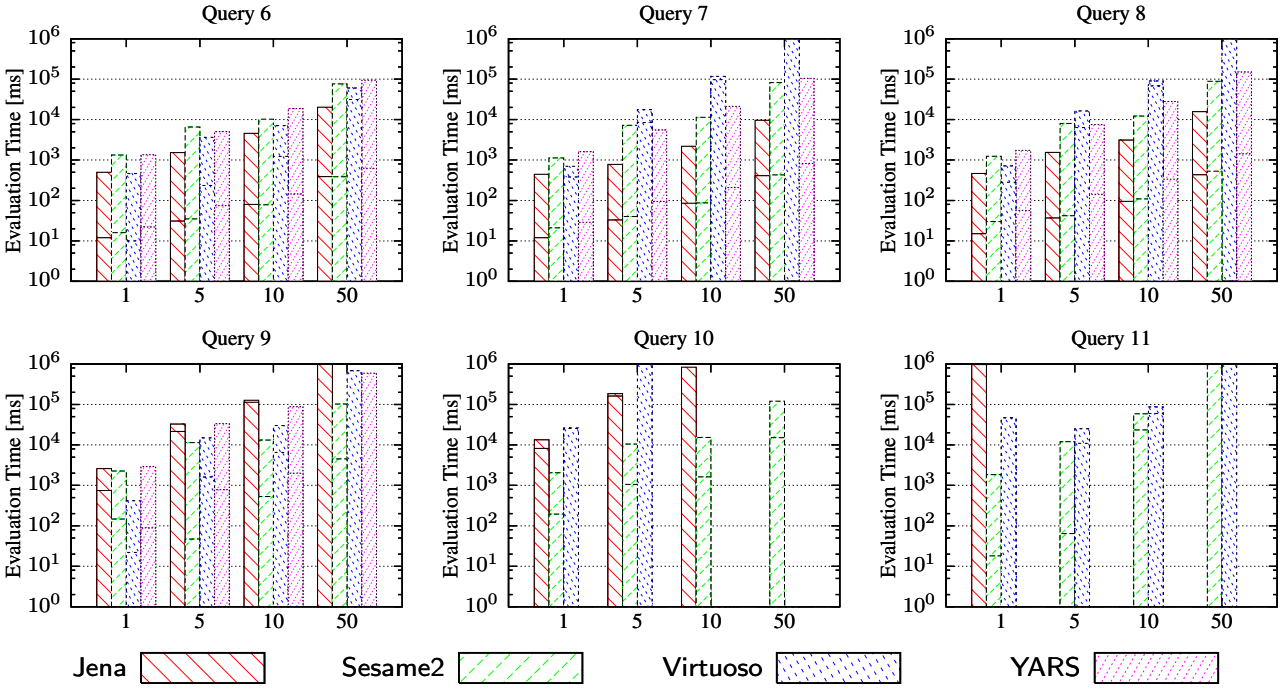


Figure 6.4: Semantic IR queries combining full-text search with structured queries.

With our semantic IR queries depicted in Figure 6.4, we test the performance of full-text search combined with structural queries. From Queries 6 to 8 an increasing amount of structural information connected to the resources that match the full-text search are to be retrieved. For Query 6, all stores have similar performance and scaling properties, but Jena always performs best. Regarding Queries 7 and 8, Virtuoso has significant scaling problems, while the performance results of all other stores are similar to those of Query 6.

With multiple keywords (Queries 9 to 11), Jena is overloaded. Virtuoso exceeds our query evaluation limit for the second smallest data set already. Sesame2 scales best with multiple keywords. Note that YARS is not capable of retrieving the correct results for queries with multiple full-text queries.

Discussion By evaluating the semantic IR queries, we recognize that Jena and Virtuoso have scalability problems for hybrid queries in general, and with those containing multiple full-text queries assigned to different resources in the query in particular. YARS also cannot properly evaluate these queries. Further investigations on Jena show that this is highly influenced by the very simple query optimizer provided by TDB. With an optimized query planning, Jena could potentially outperform Sesame2. This class of queries seems not to be considered by the evaluated RDF stores (except by Sesame2), however, it is an important class of queries in the field of semantic keyword queries.

Advanced IR Features

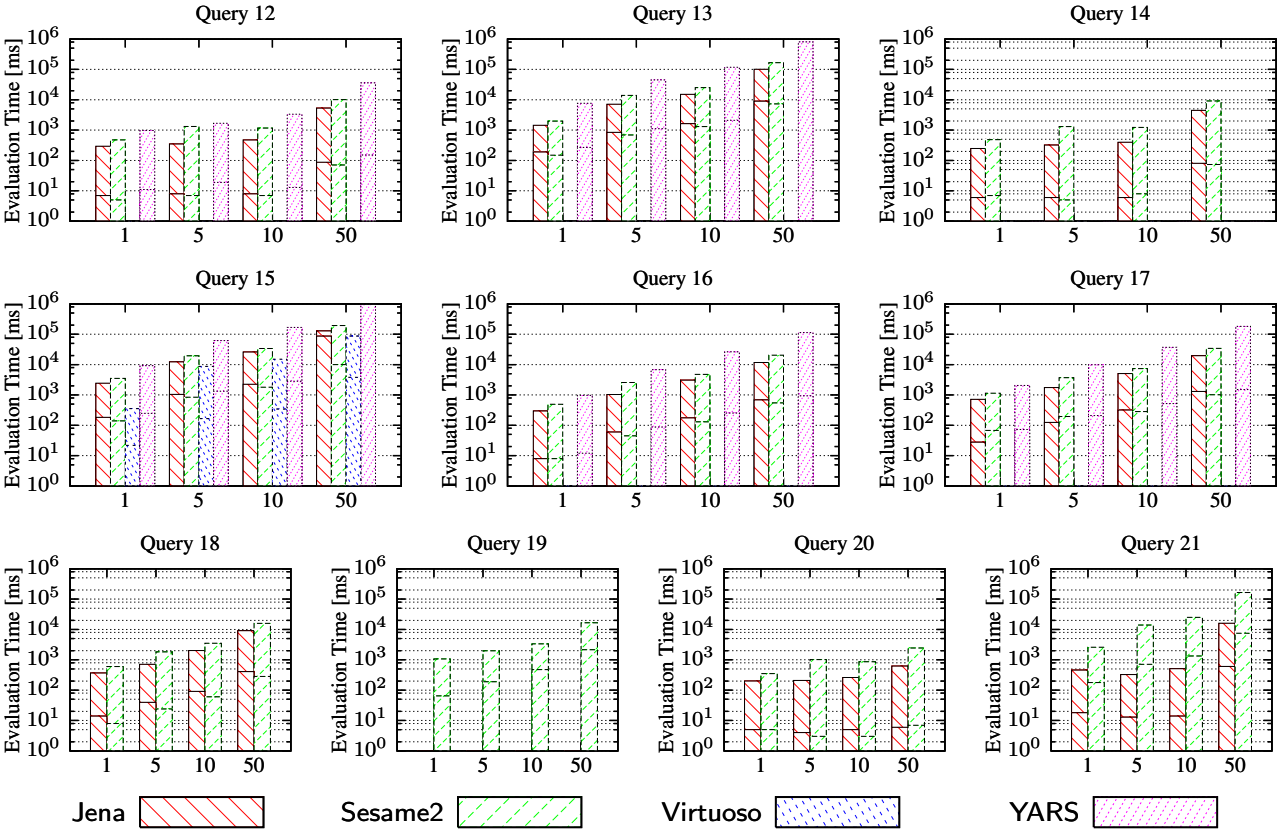


Figure 6.5: Advanced IR feature queries: boolean, proximity, wildcard, and fuzzy queries, score and snippet retrieval, result and relevance score limitation queries.

Finally, we test for the advanced IR features. Virtuoso only supports one out of ten queries, but this one with the smallest response time of all RDF stores. YARS only supports a subset of five queries. Jena and Sesame2 both have similar performance, but Jena is always faster. For Query 20 one can see that the response time is quite constant for all data sets, whereas for Query 21, this holds true only for Jena.

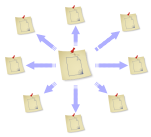
Discussion Looking at advanced IR features, Sesame2 stands out by supporting all features. For those supported by Jena, its overall performance is better. The result set limitation feature of Query 20 works perfectly for both. The evaluation time neither grows with the data set size nor with the correlating complete result set size. Limiting the results yields to an almost constant response time. Due to the fact that Sesame2 does not support the relevance score limitation feature of Query 21, this feature has to be achieved using a SPARQL FILTER. This first completely evaluates the query and then filters all results that match the score limit. Jena provides this feature with its full-text querying facility and therefore performs as expected.

Summary For basic or advanced IR queries, Jena and Sesame2 are the best choices. Considering complex semantic IR queries, Sesame2 is the only RDF store sufficiently solving the tasks of full-text search on RDF data. With ”warmed up“ caches, Virtuoso outperforms all other stores, but supports only a small subset of queries.

6.4 Conclusion

With our presented work, we addressed the strong need of application and RDF store developers for an RDF full-text benchmark. With the provided data sets and hybrid queries, full-text capabilities of any RDF store can be evaluated and compared. Our evaluation shows that on the one hand basic as well as advanced IR features are already sufficiently supported by today's RDF stores. On the other hand, it also uncovered several areas where performance needs to be improved. This holds primarily for complex hybrid queries that contain more than one full-text search condition. The evaluation results also show that a good query plan optimization is crucial for competitive performance, as well as a tight integration of full-text search features into this planning phase.

We are convinced that the availability of a standard full-text benchmark for RDF data stores will foster the development of more efficient hybrid query processing algorithms.



Chapter 7

Diversifying Search Results

Result diversification is an effective method to reduce the risk that none of the returned results satisfies a user's query intention [186, 194]. It has been shown to substantially decrease query abandonment [44, 60]. While diversification is an effective means to cater for diverse query interpretations, it is also computationally expensive. The computation of an optimally diverse result subset from a relevant input set has been shown to be NP-hard [2, 38, 79]. Therefore, existing approaches use approximation techniques and heuristics to increase the efficiency of diversification [65, 68, 79, 183, 191, 197]. While the computational complexity of state-of-the-art approximation algorithms is usually linear to the input size, they rely on random access to items of the input set. To allow for high performance, the entire input set is hold in main memory, imposing substantial memory requirements on systems processing many user queries in parallel. Further, these algorithms cannot be applied for diversification of result streams, such as results for continuous queries in databases [9, 45, 116, 142, 182], publish/subscribe systems [98, 102] or on Semantic Web data [14]. Diversification of results is impossible for systems that potentially process thousands of queries in parallel when they use algorithms with super-linear runtime and linear memory requirements.

To solve this issue, I present our streaming-based approach which processes items incrementally, maintaining a near-optimal diverse set at any point in time [126]. This approach exhibits a linear computation and constant memory complexity with respect to the size of the input set, and naturally fits for streaming applications. Further, it can be used to streamline many of the existing diversification approaches, such as MAXMIN and MAXSUM [1, 68, 69, 79]. Though memory consumption is reduced without compensation with more computation, an extensive evaluation on several real-world data sets show that diversification quality does not suffer, while efficiency is increased by orders of magnitude for very large sets as well as for continuous query scenarios such as news stream subscriptions. In particular, this work makes the following contributions:

1. It formally defines the task of *stream diversification*, which elegantly extends the usual *set diversification task*.

2. It proposes an *incremental diversification framework* suitable to diversify sets and streams.

3. Incremental approximation algorithms for the MAXSUM and MAXMIN diversification problems are provided. With N as input set size and k as output set size, the algorithms require only $O(k^2)$ memory for floating point values, $O(k)$ memory for items, $O(Nk)$ distance computations, and $O(Nk)$ and $O(Nk^2)$ other operations, respectively.

4. While substantially improving diversification efficiency, our algorithms still provide diversified sets comparable to state-of-the-art *facility dispersion* approximation algorithms [79].

This chapter is organized as follows. The next section discusses the existing work on set and stream diversification. Then, the formal foundation of diversification and our main contribution is presented, the incremental diversification framework and two incremental algorithms based on this framework. Further, their quality and performance is evaluated and compared to state-of-the-art diversification algorithms.

7.1 Related Work

Diversification is studied in many different fields such as biology, ecology, psychology, linguistics, and economics [118]. From the latter, it is known that consumer's satisfaction diverges over time when consumed things are very similar. Suchlike, Web search engine users get bored when they click through a list of very similar results, as they lose conviction that dissimilar results may follow. In 1924, this phenomenon was named "the law of diminishing returns" [176].

In media, diversity manifests itself in a number of dimensions like topics, style of writing, sentiment, and ideological perspectives [121]. In the context of Information Retrieval, these aspects should not be regarded as noise or contradictions, but rather as a rich source of information [78].

In [194], Zhai and Lafferty model the Information Retrieval task as a statistical decision problem with the aim to minimize the risk of loss. Here, loss describes how much a result fails to satisfy a user's *information need*. Since this loss is always connected to the relevance of the results, the choice of a specific loss function allows to model different approaches in result set optimization. Traditional Information Retrieval approaches are modeled by a function in which the loss is *independently* estimated for each document. Diversification can be introduced by choosing a loss function where one document depends on the other documents in the result set.

Several such diversification objectives have been proposed and successfully validated. The use of Maximum Marginal Relevance (MMR) was proposed by Carbonell and Goldstein [36], and also used in Zhai et al. [193]. Chen and Karger [44] introduced the metric k*-call at* n: this metric is 1 if at least k of the top n documents in a result set are deemed relevant, otherwise 0. The aim is to achieve a high average k-call over multiple queries. The choice of a small k leads to higher diversification, a higher k gives relevance more weight.

Gollapudi and Sharma [79] express their diversification objectives in terms of facility dispersion optimization problems. In this work, we use their MAXSUMDISPERSION and MAXMINDISPERSION algorithms as the baseline for evaluation. In contrast to [79], only the diversified set is required to be in main memory and the input items are processed incrementally. This allows to increase the efficiency for large item sets without significant loss of quality, as shown in Section 7.4. The streaming-based approach of this work is not limited to these two diversification objectives, though, and is independent of any particular relevance and distance measures.

Document similarity is not the only possible source for a diversification objective. Another established approach considers each document as covering a subset of all possible aspects of the query input [38, 53, 159, 193]. The topical similarity based on the distance in taxonomies (categorical distance) is also used for diversifying product search [79] as well as for book, bookmark, and movie recommendations [191, 197]. Also Web search engine click rates [175] and query reformulation [160] have been shown to be very valuable sources for Web search diversification.

As search engine query logs show, there is a fair number of news-related queries [128], suggesting that blog search users have an interest in the blogosphere response to news stories as these develop. To the best of our knowledge, in the context of diversification, this need has only been addressed by Drosou and Pitoura [68]. They propose algorithms that diversify a stream of results using a jumping window approach. The authors argue that an incremental computation of a diverse set of a stream is not possible. While this is true for the *optimal* diverse set, we show that an incremental heuristic approximation is very well possible. Their *SGC* algorithm — a heuristic variant based on a sliding window — is the most similar one in the literature compared to our incremental approach. However, our "window" always starts at the beginning of the stream, providing a diverse set of the entire history, rather than a fixed number of recent items. This allows our algorithm to actively withdraw items from the diverse set if this improves diversity, whereas SGC simply drops diverse items as they leave the window.

Diversification of search results is further considered in the domain of database search. A clustering and weighted sampling approach is used in [183] to obtain a diverse subset of results. Greedy algorithms are used in [87] in a similar way as presented here. However, the author does not focus design and evaluation on the streamline character of diversification, and does not consider input streams that are not ordered by relevance (*e. g.*, from continuous queries).

7.2 Diversification Problems

In this section, I formally define the problem of diversification and provide the notation for the rest of this chapter, mostly following [79]. I first define the classical *Set Diversification* problem and then extend this notion to the problem of *Stream Diversification* to cover diversification of continuous data.

Set Diversification For a given set of results $\mathbf{U} : |\mathbf{U}| = N$ (*e. g.*, search results, news or blog articles matching a user query) one aims to find the set $\mathbf{S}_k \subseteq \mathbf{U} : |\mathbf{S}_k| = k$ that is both relevant *and* diverse. Relevance of a result measures the probability that the user is actually interested in that result ($w : \mathbf{U} \to \mathbb{R}^+$)[1], whereas diversity reflects dissimilarity ((1 - similarity), also denoted distance) of one result to another ($d : \mathbf{U} \times \mathbf{U} \to \mathbb{R}^+$). To simplify the presentation, I assume in the following that d is a metric. Usually, relevance and diversity are conflicting objectives: when the top-k most relevant results do not constitute the maximal diverse set, diversity can only be increased by replacing a top-k result with a less relevant but more dissimilar one. This obviously reduces the overall relevance of the set. The trade-off between the two measures is captured in a *diversification objective*, a function over a set of items that consolidates the relevance of results $w(\cdot)$ and the diversity between pairs of results $d(\cdot, \cdot)$ into a single measure:

$$f : 2^{\mathbf{U}} \times w \times d \to \mathbb{R} \tag{7.1}$$

where $2^{\mathbf{U}}$ denotes the power set of $\mathbf{U}$.

[1] While the result set $\mathbf{U}$ as well as the relevance function w depend on the user query, the explicit introduction of the query itself into this diversification model would complicate the notation (*cf.* [79]).

Having the diversification objective at hand, the *diversification problem* can be defined as finding the k-size random subset $\mathbf{S}_k \in 2^{\mathbf{U}}$ that maximizes the diversification objective:

$$\mathbf{S}_k^* = \arg\max_{\substack{\mathbf{S}_k \in 2^{\mathbf{U}} \\ |\mathbf{S}_k|=k}} f(\mathbf{S}_k, w(\cdot), d(\cdot,\cdot)) \tag{7.2}$$

This optimization is known to be NP-hard [2, 79]. As the number of candidate sets $|\{\mathbf{S}_k \in 2^{\mathbf{U}} : |\mathbf{S}_k| = k\}|$ is $\frac{N!}{(N-k)!}$, an exhaustive exploration of all sets to find the optimal solution is impractical even for small $k \approx 10$ as soon as $N > 20$. Therefore, all proposed diversification approaches use greedy algorithms and/or heuristics to approximate the optimal solution $\mathbf{S}_k^*$ by a near-optimal $\mathbf{S}_k^+$.

Instances of Diversification Objectives and Problems Throughout the literature of diversification, there are two popular diversification objectives: SUM[2] and MIN [68, 79]. As the names imply, they are based on the sum or minimum of the two measures w and d. Formally, the former is defined as

$$f_{\text{SUM}} = (k-1)\sum_{i \in \mathbf{S}_k} w(i) + \lambda \sum_{\substack{i,j \in \mathbf{S}_k \\ i \neq j}} d(i,j) \tag{7.3}$$

The first term in Equation 7.3 sums the relevance scores for each result, the second term sums the distances of all pairs $(i,j) \in \mathbf{S}_k \times \mathbf{S}_k : i \neq j$. Since the first sum runs over k values and the second over $k(k-1)$, this imbalance is compensated by multiplying the first term with $k-1$. In case of $d(\cdot,\cdot)$ being a metric [79, Eq. 1], it suffices to sum over all $\frac{k(k-1)}{2}$ unique unordered pairs and multiply the second term by 2.

The parameter λ allows to balance the influence of relevance and diversity on the objective value. A $\lambda < 1.0$ increases the contribution of relevance, whereas a $\lambda > 1.0$ favors item dissimilarity. In order to consider both cases, $\lambda = 0.5, 1.0, 2.0$ is used for the evaluation.

The MIN objective is defined as

$$f_{\text{MIN}} = \min_{i \in \mathbf{S}_k} w(i) + \lambda \min_{\substack{i,j \in \mathbf{S}_k \\ i \neq j}} d(i,j) \tag{7.4}$$

where only minimum relevance and minimum pairwise distance contribute to the objective.

The diversification problems of maximizing the SUM and MIN objectives are referred to as MAXSUM and MAXMIN[3]. These problems can be cast as facility dispersion problems as shown in [79], for which approximation algorithms exist. These algorithms are briefly summarized in the following.

[2]The AVG objective bases on *average* relevance and *average* pairwise distance. For a fixed k, it is proportional to the SUM objective and is therefore not separately considered in this work.

[3]This work specifically distinguish between diversification objectives (SUM and MIN) and their respective diversification problems (MAXSUM and MAXMIN) that aim at maximizing the objective.

The MAXSUMDISPERSION algorithm composes the diverse set by selecting $\frac{k}{2}$ pairs of most distant items, with distance defined as $d(i,j)' = w(i)+w(j)+2\lambda d(i,j)$. If k is odd, the k^{th} item is randomly added to the set. Finding the pair of most distant items is of time complexity $O(N^2)$, thus this algorithm belongs to the complexity class $O(N^2k)$. For certain distance functions, this complexity can be reduced by employing specialized index structures or computations [1].

Similarly, MAXMINDISPERSION initializes the diverse set with the pair of most distant items, where distance is defined as $d'(i,j) = \frac{1}{2}(w(i) + w(j)) + \lambda d(i,j)$. Then, it adds the item i that maximizes $\min_{j \in \mathbf{S}} d'(i,j)$. After $k-2$ iterations, the diverse set is constructed. Again, finding the pair of most distant items requires $O(N^2)$ distance computations, while the $k-2$ iterations demand $O(Nk)$ operations, summing up to $O(N^2 + Nk)$. As described in [68], the MAXMINDISPERSION algorithm can be improved by the following heuristic: instead of selecting the initial two most distant items from the whole input set, it can selected from a uniformly random sample set of size h ($h \ll N$). This has only a small impact on the diversification quality [67], but reduces the complexity to $O(Nk + h^2)$.

Since both algorithms require random access to the input set, it is usually held in memory, yielding to memory complexity of $O(N)$. This makes the application of these algorithms practically infeasible for large sets. In case of data streams, the algorithms need to be executed repeatedly as new items arrive, causing an even higher computational effort. In the following Section 7.3, I present our incremental approach that solves these issues.

Stream Diversification As an extension to the *set diversification problem*, let's consider the items of the set to be a sequence or *stream* $\mathbf{S} = (s_1, \dots)$ in the order as retrieved by the respective IR system. For a given query, a traditional document retrieval system provides documents in descending order of relevance. A news or blogs publish/subscribe system provides results matching a continuous query as they are recognized by the system, *i. e.*, ordered by time. The diversification problem over a stream $\mathbf{S}$ at each position i in the stream is defined as

$$\mathbf{S}^*_{k,i} = \arg\max_{\substack{\mathbf{S}_k \in 2^{\mathbf{S}_i} \\ |\mathbf{S}_k|=k}} f(\mathbf{S}_k, w(\cdot), d(\cdot,\cdot)) \tag{7.5}$$

where $\mathbf{S}_i = \{s_j : j \leq i\}$ is the set of the items at positions $[1,i]$. In other words, for each position i in $\mathbf{S}$, a diversified set $\mathbf{S}^*_{k,i}$ is defined over the subsequence $\mathbf{S}_i = (s_1, s_2, \dots, s_i)$. It can be seen that the set diversification problem is a special case of stream diversification, by viewing the input set $\mathbf{U}$ as stream $\mathbf{S} = (s_1, \dots, s_N)$. Then,

$$\mathbf{S}^*_k \overset{\mathbf{U}=\mathbf{S}_N}{=} \mathbf{S}^*_{k,N} \tag{7.6}$$

In contrast to our stream diversification problem, [68, 69] consider a fixed window of recent items for the diversification task. This, for instance, poses a problem for items like news articles or blogs that are not published at a fixed rate. Then, at two different points in time the window covers different periods of time. Further, picking the window size represents an additional optimization problem.

Diversification along a time-ordered stream of results may lead to diversified k-size sets that primarily contain "old" but relevant and highly diverse results while new, similar results are omitted. Such a behavior can be compensated by introducing a decay factor to the relevance function which lowers the item score as its age increases [68]: Function $w_i(s_j)$ is the relevance of item s_j at time i (i.e., when item s_i is observed):

$$w_i(s_j) = w(s_j) * e^{-\alpha t(s_i, s_j)} \tag{7.7}$$

with $\alpha = -\frac{\log 0.5}{\log e} t_{1/2}^{-1}$ and $t_{1/2}$ being the half-life time of results' relevances, while $t(s_j, s_i) = \max(t(s_i) - t(s_j), 0)$ provides the time difference between result s_i and s_j if the point in time $t(s_i) > t(s_j)$, or 0 otherwise.

7.3 Incremental Diversification

The following section describes our *incremental diversification framework*. It is composed of a generic *incremental diversification algorithm* that employs a general *incremental objective model* to optimize computation. This algorithm efficiently and effectively produces diverse k-size sets in linear time (for fixed k) and over streams in constant time (at each position) with respect to input size. Based on this framework, I develop incremental algorithms for the MAXSUM and MAXMIN diversification objectives.

7.3.1 Incremental Diversification Framework

Incremental Diversification Algorithm The core idea of the incremental diversification algorithm is to process the input as a stream of items and to continuously maintain a diverse subset at each position of the stream. Let $\mathbf{S}^+_{k,i-1}$ be a near-optimal diverse k-element subset of $(s_1, \ldots, s_{i-1})$. Then, its successor $\mathbf{S}^+_{k,i}$ is determined as the set with the highest objective among all candidate sets $\mathbf{S}'_{k,i}$ that can be created by replacing at most one element of $\mathbf{S}^+_{k,i-1}$ with s_i.

Formally, this can be defined as

$$\mathbf{S}^+_{k,i} = \underset{\substack{\mathbf{S}'_{k,i} \in 2^{\mathbf{S}^+_{k,i-1} \cup \{s_i\}} \\ |\mathbf{S}'_{k,i}| = k}}{\arg\max} f(\mathbf{S}'_{k,i}, w(\cdot), d(\cdot,\cdot)) \tag{7.8}$$

Put differently, two consecutive sets $\mathbf{S}^+_{k,i-1}$ and $\mathbf{S}^+_{k,i}$ with $i \in [k+1, N]$ relate to each other as

$$\mathbf{S}^+_{k,i} = \mathbf{S}^+_{k,i-1} \setminus \{s_j\} \cup \{s_i\} : s_j \in \mathbf{S}^+_{k,i-1} \wedge s_j \notin \mathbf{S}^+_{k,i} \tag{7.9}$$

if a modification took place, otherwise $\mathbf{S}^+_{k,i} = \mathbf{S}^+_{k,i-1}$. For the first k items, $\mathbf{S}^+_{k,i}$ is trivially defined as $\mathbf{S}^+_{k,i} = \mathbf{S}^+_{k,i-1} \cup \{s_i\} : i \leq k$ with $\mathbf{S}_{k,0} = \emptyset$.

Algorithm 5 specifies this approach in pseudo-code. At the end of each iteration i over $[1, N]$, the algorithm provides the diverse set $\mathbf{S}^+_{k,i}$. In case of streams, this algorithm can continue com-

putation as soon as a new item arrives.

Additionally, this algorithm can be modified in line 10 to minimize the objective value, rather than maximizing it. This allows to also use the algorithm along with all those diversification objectives that are to be minimized, for instance the MAXCOV objective [174].

An important characteristic of this algorithm is that the objective value is computed in the inner loop for k candidate sets $S'_{k,i}$ that, compared to the outcome set of the previous iteration, mutually differ in only one item. In the following, we show how the objective computation can further be optimized by exploiting this fact.

Incremental Objective Model The objective value of a set $\mathbf{S}^+_{k,i}$ is computed based on the known objective value of $\mathbf{S}^+_{k,i-1}$. Since both sets relate to each other as defined by Equation 7.9, only the relevance scores of item s_j (removal) and s_i (addition), as well as the distances $d(s_j, s_l)$ and $d(s_i, s_l) : s_l \in \mathbf{S}^+_{k,i-1} \setminus \{s_j\}$ need to be considered for computation. The incremental diversification algorithm computes the objective of different $\mathbf{S}^+_{k,i}$ by subsequently varying s_j (denoted in line 8 as $\mathbf{S}'_{k,i}$). Therefore, each distance $d(s_i, s_l)$ is reused $k-1$ times in the inner loop and does not need to be recomputed.

Algorithm 5: Incremental Diversification Algorithm

Input: Set of items $\{s_1, \ldots, s_N\}$, size of diverse set k
Output: diverse set of items $\mathbf{S}^+_{k,N}$

```
1  for i ← 1 to N do
       // The first k items are added to the diverse set
2      if i ≤ k then
3          S+_{k,i} ← S+_{k,i-1} ∪ {s_i} ;
4          next ;
       // Memorize objective of current diverse set
5      max ← f(S+_{k,i-1}, w(·), d(·,·)) ;
       // Memorize current diverse set
6      S+_{k,i} ← S+_{k,i-1} ;
       // Try to replace each element of current diverse set
7      for ∀s_j ∈ S+_{k,i-1} do
           // Construct next candidate set
8          S'_{k,i} ← S+_{k,i-1} \ {s_j} ∪ {s_i} ;
           // Check If new objective is larger, and memorize if so
9          current ← f(S'_{k,i}, w(·), d(·,·)) ;
10         if current > max then
11             max ← current ;
12             S+_{k,i} ← S'_{k,i} ;
13 return S+_{k,N} ;
```

7.3.2 Incremental SUM Objective

The core idea of the incremental SUM objective function exploits the fact that a sum $s = \sum_{i=1}^{n} a_i$ can — instead of recomputing it — be updated when one value $a_j : j \in [1, n]$ gets replaced by a' as $s' = s + a' - a_j$. Equivalently, f_{SUM} in Equation 7.3 with symmetric distance can be incrementally computed as:

$$f_{\text{SUM}}(\mathbf{S}'_{k,i}) = f_{\text{SUM}}(\mathbf{S}^{+}_{k,i-1}) + (k-1)(w(s_i) - w(s_j)) + 2\lambda \sum_{\substack{s_l \in \mathbf{S}^{+}_{k,i-1} \\ s_l \neq s_j}} (d(s_i, s_l) - d(s_j, s_l)) \tag{7.10}$$

where $f_{\text{SUM}}(\mathbf{S})$ is a short hand notation for $f_{\text{SUM}}(\mathbf{S}, w(\cdot), d(\cdot,\cdot))$.

As the incremental algorithm computes the objective of k different candidate sets $\mathbf{S}'_{k,i}$ with only s_j varying, the algorithm keeps all distances between the elements of $\mathbf{S}^{+}_{k,i-1}$ in a matrix $\mathbf{D} = (d_{m,n})$, where $d_{m,n} = d(s(m), s(n)) : \forall n, m = 1, \ldots, k$. The function $s(\cdot)$ provides the element of the diverse set that corresponds to the index $[1, k]$ of this matrix and the following vectors. The symmetry of $d(\cdot)$ leads to a symmetric matrix $\mathbf{D}$. During the construction of the initial set (line 3 of the algorithm), this matrix is built up for the first k items. From then on, it is updated after the inner loop, as shown later.

The distance between the new item s_i and the items in the current diverse subset, $d(s_i, s(l)) : l = 1, \ldots, k$, is reused $k - 1$ times for each $s(l)$. Therefore vector $\mathbf{d} = (d_1, \ldots, d_k)$ is defined with $d_l = d(s_i, s(l)) : l = 1, \ldots, k$. This vector is computed once before the inner loop. Further, the sum over all these distances is defined as $\delta_{k,i} = \sum_{l=1}^{k} d_l$.

The vector $\varsigma_{k,i-1} = (\sigma_{1,i-1}, \ldots, \sigma_{k,i-1})$ with $\sigma_{m,i-1} = \sum_{n=1}^{k} d_{m,n} : m = 1, \ldots, k$ provides the row sums of matrix $\mathbf{D}$.

Then, Equation 7.10 can be rewritten as

$$\begin{aligned} f_{\text{SUM}}(\mathbf{S}'_{k,i}) = f_{\text{SUM}}(\mathbf{S}^{+}_{k,i-1}) \quad & +(k-1)(w(s_i) - w(s_j)) \\ & +2\lambda(\delta_{k,i} - d_j - \sigma_{j,i-1}) \end{aligned} \tag{7.11}$$

In case $\mathbf{S}^{+}_{k,i}$ differs from $\mathbf{S}^{+}_{k,i-1}$ (after the inner loop), the following updates are required: Vector $\varsigma_{k,i}$ is derived from $\varsigma_{k,i-1}$ for all $l = 1, \ldots, k$ as

$$\sigma_{l,i} = \begin{cases} \sigma_{l,i-1} + d_l - d_{j,l} & : l \neq j \\ \delta_{k,i} - d_l & : l = j \end{cases} \tag{7.12}$$

where j refers to the index of s_j as $s(j) = s_j$. Then, matrix $\mathbf{D}$ is updated as $d_{l,j} = d_{j,l} = d_l : l = 1, \ldots, k$, and function $s(\cdot)$ is modified so that $s(j) = s_i$. With this strategy, our incremental SUM objective achieves a computation complexity of $O(1)$, while requiring $O(k^2)$ memory.

7.3.3 Incremental MIN Objective

The MIN objective defined in Equation 7.4 bases on the minimum relevance and pairwise distance. Therefore, the objective of each $\mathbf{S}'_{k,i}$ is computed with the new minimum relevance and distance values of $\mathbf{S}^+_{k,i-1}$ and item s_i. Similar to the incremental SUM objective, the pairwise distances between s_i and $s_j \in \mathbf{S}^+_{k,i-1}$ are computed once before the outer loop of the incremental algorithm and reused to compute the objective of the candidate sets:

$$\min_{s_n \in \mathbf{S}'_{k,i}} w(s_n) = \min \left(\left(\min_{\substack{s_n \in \mathbf{S}^+_{k,i-1} \\ s_n \neq s_j}} w(s_n) \right), w(s_i) \right) \tag{7.13}$$

$$\min_{\substack{s_n, s_m \in \mathbf{S}'_{k,i} \\ s_n \neq s_m}} d(s_n, s_m) = \min \left(\left(\min_{\substack{n,m \in \{1,\dots,k\} \setminus \{j\} \\ n \neq m}} d_{n,m} \right), \min_{n \in \{1,\dots,k\} \setminus \{j\}} d_n \right) \tag{7.14}$$

With an ordered vector of all relevances of the diverse set, Equation 7.13 can be computed in $O(1)$ by comparing the first element (if the respective relevance does not belong to s_j) or the second element (otherwise) of the vector with $w(s_i)$. Equation 7.14 can be evaluated by finding the smallest distance among $d_{n,m}$ and d_l with $n, m, l \in [1, k]$ that does not belong to s_j. This can be done by a *sort-join* [75, Sec. 15.4.5] or *two-way merge* [110, Sec. 5.2.4] over the ordered vector of all $d_{n,m}$, the ordered vector of all d_l, and the ordered vector of $d_{j,m}$. The result distance is the minimum of the former two vectors that does not appear in the latter one (thus it does not belong to s_j). This operation has a complexity of $O(k)$ in worst-case when the k smallest pairwise distances of $\mathbf{S}^+_{k,i-1}$ belong to s_j and all distances to s_i are larger than these. For the best case, in which the smallest pairwise distance of $\mathbf{S}^+_{k,i-1}$ does not belong to s_j and is larger than the smallest d_l, Equation 7.14 has a complexity of $O(1)$.

7.3.4 Complexity Analysis

The complexities of the incremental objectives for MAXSUM and MAXMIN are briefly outlined before. I now analyze the total complexity of our algorithm in detail. The complexities of all algorithms are summarized in Table 7.1 for condensed reference.

The computation and memory complexities are each considered to be compound. The computation contains distance computations and basic arithmetic operations. The former may involve several of the latter, potentially rendering distance computations to be of an order of magnitude slower. Memory consumption can be considered as a conjunction of the memory that stores the items (to be diversified) and memory for basic data types such as double precision floating point values. Again, the former can be orders of magnitude larger than the latter.

As one can see from the pseudo-code, the incremental diversification algorithm has a complexity of $O(Nk)$ set modifications and objective computations.

In case of MAXSUMINCREMENTAL, the structures $\mathbf{D}$, $\mathbf{d}$, $\sigma_{j,i-1}$, and $\delta_{k,i}$ are constructed and updated incrementally outside of the inner loop, requiring $O(k)$ operations for each item. As shown, the computation of the objective itself is $O(1)$. Thus, the total computational complexity of MAXSUMINCREMENTAL is still $O(Nk)$.

For MAXMININCREMENTAL, the objective computation has a complexity of $O(k)$ in the worst-case and $O(1)$ in best case. The removal and addition of elements from and to the ordered relevance and distances vectors each requires $O(\log k)$ operations. These operations are performed before and after the inner loop, leading to a total complexity of $O(Nk^2)$ and $O(Nk \log k)$, respectively. For both objectives, the memory requirements for maintaining the distance matrix dominates the overall memory complexity, leading to $O(k^2)$. Note that k is usually rather small, according to the "less is more" [44] motto. Thus, our algorithm is essentially linear in the number of items, with a small constant factor determined by k.

When exponential decay is part of the relevance function, our incremental algorithms have to recompute the relevance scores at each point in time. This is done in the outer loop with cost $O(k)$ and does not affect the overall complexity.

diversification algorithm	acronym	CPU		Memory	
		distances	basic	items	basic
MAXSUMDISPERSION	`MSDisp`	$O(N^2k)$	$O(N^2k)$	$O(N)$	$O(1)$
MAXSUMDISPERSION cached	`MSDisp+`	$O(N^2)$	$O(N^2k)$	$O(N)$	$O(N^2)$
MAXSUMINCREMENTAL	`MSInc`	$O(Nk)$	$O(Nk)$	$O(k)$	$O(k^2)$
MAXMINDISPERSION	`MMDisp`	$O(N^2 + Nk)$	$O(Nk)$	$O(N)$	$O(1)$
MAXMINDISPERSION cached	`MMDisp+`	$O(N^2)$	$O(Nk)$	$O(N)$	$O(N^2)$
MAXMINDISPERSION cached (h=2)	`MMDisp+h2`	$O(Nk + h^2)$	$O(Nk)$	$O(N)$	$O(Nk + h^2)$
MAXMININCREMENTAL	`MMInc`	$O(Nk)$	$O(Nk^2)$	$O(k)$	$O(k^2)$

Table 7.1: Complexity in CPU consumption (distance computation and basic operations) and memory consumption (items and basic data types) of different diversification algorithms. Basic operations are arithmetic operations or memory access, basic data types are double precision floating point values.

7.4 Experimental Evaluation

I presented a generic incremental diversification framework, as well as two specific incremental algorithms, MAXMININCREMENTAL and MAXSUMINCREMENTAL. As shown, computation and memory complexity is reduced compared to their *baseline* algorithms MAXMINDISPERSION and MAXSUMDISPERSION, at the cost of giving up approximation guarantees. Therefore, the main goal of the experimental evaluation is to show that for real-world data, this does not lead to a decrease in diversification quality. In addition, the performances of the algorithms and the baseline are measured to confirm the analytical results presented in Section 7.3.4. Finally, effectiveness and efficiency of the algorithms are investigated for stream diversification.

7.4.1 Experimental Setup

Data Sets and Queries To show the wide applicability of the incremental algorithm, they are evaluated on three different well-known data sets. We opted for data sets that provide large result sets and streams — the challenge we tackle with our incremental diversification algorithms. In the following, I describe the data sets and corresponding queries used for evaluation.

NYTimes The *New York Times Annotated Corpus* [158] contains over 1.8 million articles of the New York Times spanning more than twenty years from 1987 until 2007. We obtain over 5,000 user queries from the AOL query log that guide users to `query.nytimes.com` and `topics.nytimes.com`. The publication timestamp of publication renders this data set particularly suitable to generate time-ordered result streams from queries.

Blogs08 The *TREC Blogs 2008 data set*[4] provides over 28 million multi-language blog entries crawled roughly over the period of the year 2008. We consider a subset of 18 million entries that can be classified[5] as English. As queries, we use the TREC 2008 Blog Track topics of the *opinion search and polarity task*[6]. From these 150 topics, we take the title as a keyword query. Again, the dates of blog retrieval allow us to generate time-ordered streams of blog entries relevant to user queries.

DBpedia The *DBpedia data set v3.5.1*[7] is a semantic data set providing the knowledge of the Wikipedia project as semi-structured data. We use DBpedia as complement to the other two document-centric data sets. Structured result sets and streams may exhibit different characteristics and challenge diversification algorithms differently than unstructured document collection. This data set contains almost 89 million facts extracted from Wikipedia content.

For the DBpedia data set, we aim at generating structured queries that most likely produce long result sets suitable for our evaluation purposes and are likely to be issued. Therefore, we exploit frequent structures exposed by the data. From the most frequent types (classes), links between these types, and their most frequent attributes (literal properties), we automatically generate queries containing at most five interconnected typed nodes (resources) and multiple optional unbound attributes. With that approach, we create over 800 queries.

These three data sets are used to generate realistic result sets and streams to put our diversification algorithms under real workload. With a result set ordered by relevance we face the *set diversification* task, ordered by time we tackle the *stream diversification* task.

[4] TREC Web Corpus — BLOGS08: `http://ir.dcs.gla.ac.uk/test_collections/blogs08info.html`.
[5] TextCat — Java Text Categorizing Library: `http://textcat.sourceforge.net/`.
[6] Topics for opinion search and polarity task (topics 851–950, 1000–1050): `http://trec.nist.gov/data/blog08.html`.
[7] DBpedia: `http://dbpedia.org/`.

Relevance and Distance Functions The diversification tasks we are focusing on require relevance and distance functions, which are to be defined for each type of data set individually.

NYTimes/Blogs08 The two document-centric data sets are indexed with Java Lucene[8] along with the Snowball stemmer[9]. We use Lucene's normalized relevance score as relevance function. Based on the cosine similarity $cosine\ similarity(\cdot,\cdot)$ of the term frequency vectors, the distance between items is defined as $d(s_i, s_j) = 1 - cosine\ similarity(s_i, s_j)$ (see page 72 in Section 5.2).

DBpedia The semantic data set is indexed and queried using the Java Sesame[10] library. After querying, results that refer to the same resources but with varying values for their attributes are consolidated into a single result with multi-value attributes. This provides the user with the full spectrum of resource attributes' values while presenting fewer results.

Following [157], we define the distance between two results s_i and s_j as a multi-type-multi-value distance measure:

$$d(s_i, s_j) = \frac{1}{n} \sum_{l=1}^{n} d_l(s_i(a_l), s_j(a_l)) \tag{7.15}$$

where the query contains the attributes $\mathbf{a} = (a_1, \ldots, a_n)$ and $s_i(a_l)$ refers to the multi-value for attribute a_l of result s_i. Further, the distance $d_l(\mathbf{v_i}, \mathbf{v_j})$ between multi-value $\mathbf{v_i}$ and $\mathbf{v_j}$ of result s_i and s_j for attribute a_l is defined as

$$d_l(\mathbf{v_i} = s_i(a_l), \mathbf{v_j} = s_j(a_l)) = \frac{1}{|\mathbf{v_i}| \cdot |\mathbf{v_j}|} \sum_{t \in \mathbf{v_i}} \sum_{u \in \mathbf{v_j}} d_l(t, u) \tag{7.16}$$

which corresponds to the average pairwise distance between the values of result s_i and s_j for attribute a_l. The distance between empty multi-values is defined as 1. Obviously, this definition considers two sparse results with disjoint sets of non-empty attributes to be maximally distant (diverse). Consequently, the diversification would favor sets of sparse results, which is unlikely to satisfy users. Therefore, we define relevance as the fraction that reflects the number of attributes of the query that are covered by the result (non-empty). A result that provides values for all attributes obtains relevance 1, whereas results with only a few attribute values get assigned a relevance close to 0. The diversification objective that balances diversity and relevance also balances between sparse and non-sparse results.

The result distance measure can employ different distance measures for different attributes. For this evaluation, cosine similarity is used over term frequency vectors for textual values and the absolute difference between numerical values, normalized by the maximum range of the respective attribute.

[8] Apache Lucene: `http://lucene.apache.org/java/`.
[9] Snowball Stemmer: `http://snowball.tartarus.org/`.
[10] OpenRDF Sesame: `http://www.openrdf.org/`.

Evaluation approach In our evaluation we target at effectiveness and efficiency of our incremental diversification algorithms compared to the baseline algorithms.

Effectiveness The primary challenge to evaluate diversification approaches is to measure result set diversity. Clearly, this measurement must be independent of the diversification objective, which is maximized. A number of established diversity-aware evaluation measures for search approaches exist, e.g., α-*NDCG* [53], *NDCG-IA* [2], and *S-recall* [193]. These measures also allow for comparison of different diversification approaches and objectives.

These measures require sub-topic annotations or relevance judgments which are not available for result set sizes in the order of 10^5 that we evaluate. In fact, this is not necessary, as we do not evaluate the quality of diversification itself, but rather how well our incremental algorithms perform compared to already established approximations [79]. Therefore, a direct comparison of the achieved objective scores instead of using one of the above mentioned measures is appropriate to assess the effectiveness of our approach.

For comparison, we run each algorithm for all queries, and measure the objective score of the diverse sets. These scores are then normalized to the interval $[0, 1]$ by division by the maximal possible objective value. The difference of an algorithm's and its baseline's normalized objectives renders the *absolute objective difference* (AOD) measure. These are averaged over all queries and used to quantify the loss of quality introduced by streamlining. An AOD value of ± 0.1 expresses an improvement / impairment of 10% within the normalized interval of $[0, 1]$.

Efficiency The algorithms efficiency is measured as follows. The queries are evaluated against the data sets and all results are fetched into memory. Then, the algorithms perform the diversification on these in-memory result sets. We only measure the time required for the diversification itself, not taking into account query execution and fetching of the initial result set into memory. The experiments employed the high performance computing cluster with Java™ version 1.6.0_16, employing Intel® Xeon® (X5670, 2.93GHz) processors. We measure the actual CPU time consumed by the algorithms, not the "wall clock" time.

Computing the baseline for our experiments are too expensive for very large result sets. Therefore, for MAXSUMDISPERSION we only consider result sets of up to 4,000 elements, for MAXMINDISPERSION up to 10,000 elements.

7.4.2 Set Diversification

Algorithms For each of the two objectives that we evaluate here, a number of approximation algorithms and variants exist. As baseline algorithms we consider the facility dispersion-based algorithms MAXSUMDISPERSION and MAXMINDISPERSION [79]. In the following, we refer to these as `MSDisp` and `MMDisp`. Both can be modified to trade fewer computations against higher memory consumption by caching computed distances. We refer to these as `MSDisp+` and `MMDisp+`, respectively. Further, the variant of `MMDisp+` that initially selects the most distant pair from a h-size random subset of all N items ($h \ll N$) is also evaluated as `MMDisp+h2` with $h = 2$. Our incremental algorithms are in the following referred to as `MSInc` and `MMInc`. As *reference algorithms*, we also show the diversity level reached by returning the top-k most relevant item, and the diversity of a k-random selection. We call these `Top` and `Rand`, respectively.

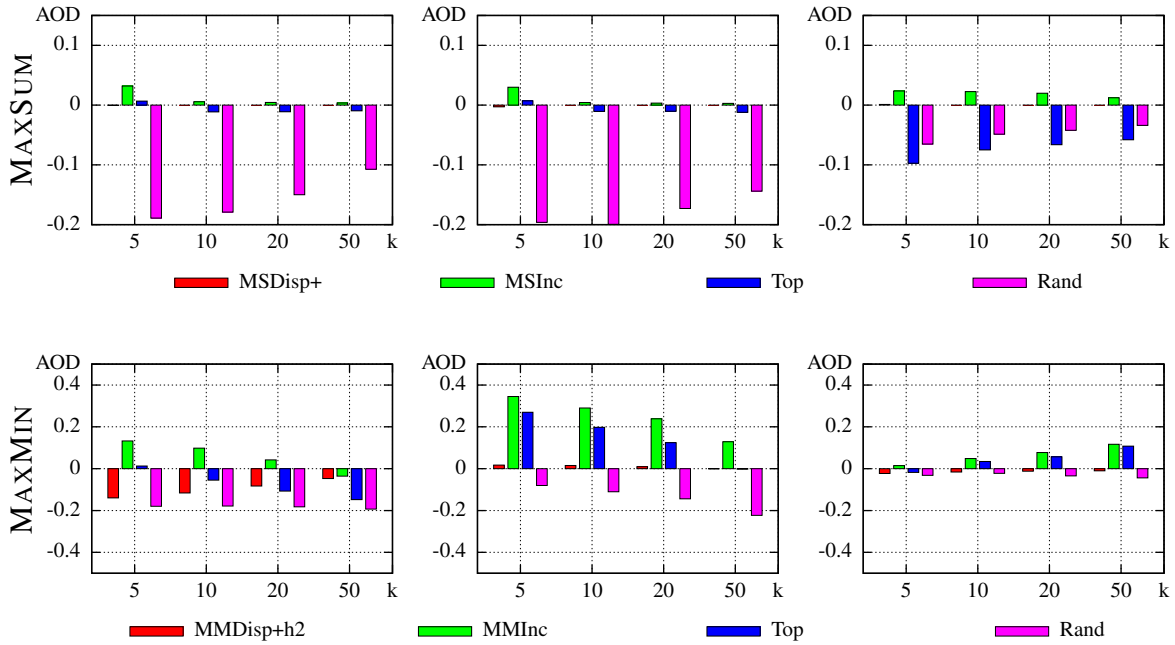


Figure 7.1: Diversification quality compared to baseline MAXSUMDISPERSION (top row) and MAXMINDISPERSION (bottom row) on the NYTimes, Blogs08 and DBpedia data set (left to right column), with $\lambda = 1$.

Diversification Quality Figure 7.1 shows the difference of the objective scores achieved with respect to the baseline algorithm, *i. e.*, the baseline algorithm's score is at 0. For the upper row with the SUM objective, the baseline algorithm is `MSDisp`. The lower row shows results for the MIN objective with `MMDisp` as baseline. `MSDisp+` shows the same diversification quality as the baseline, because the only difference is that the former caches results.

For all three data collections, `MSInc` clearly is on par with the baseline. In some instances, the incremental results are even better. Our algorithms *can* be better than the baseline because both approximate the optimal solution. `MSInc` *is* sometimes better because it considers every item in the set, while `MSDisp` only considers items with maximal distance within the remaining item set (see Algorithm 1 in [79]). For instance, two identical pairs of results (a_1, b_1) and (a_2, b_2) with maximal distance $d(a_1, b_1) = d(a_2, b_2)$ are both added to the diverse set, though $d(a_1, a_2) = d(b_1, b_2) = 0$ does not contribute diversity.

For the MIN objective, we observe a higher variation of the AOD values. However, also for this objective, the incremental algorithm yields similar or better results than the baseline.

`MSInc` results are more stable over all data sets and configurations than `MMInc`. This characteristic is due to the fact that for MIN the item with the smallest relevance and distance determines the objective score of the entire set, whereas for SUM all items contribute to the final score.

Though `MMDisp+h2` yields reasonable results for Blogs08 and DBPedia, this approach does not work well for the NYTimes collection (Figure 7.1, bottom left).

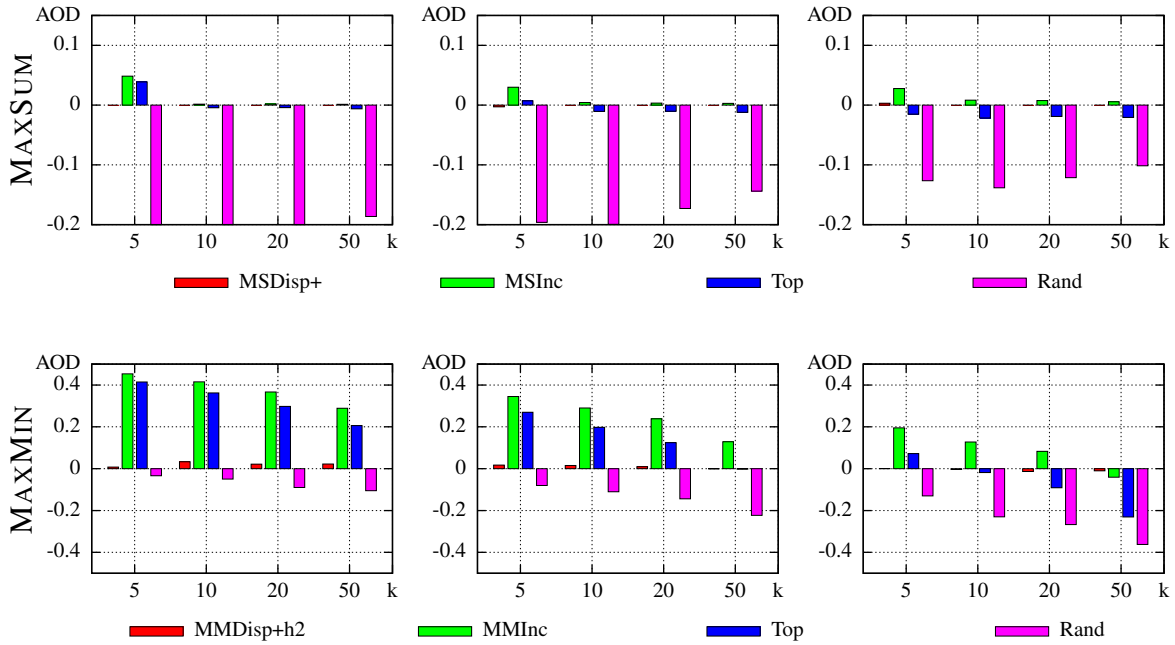


Figure 7.2: Diversification quality on the Blogs08 data set compared to baseline algorithm MAXSUMDISPERSION (top row) and MAXMINDISPERSION (bottom row) for $\lambda = 0.5, 1.0, 2.0$ (left to right column).

Influence of λ To investigate the influence of different trade-offs between relevance and distance, we now vary λ between 0.5 and 2.0. The results are shown in Figure 7.2. For SUM, the approximation quality is not substantially influenced by the choice of λ. On the other hand, the respective differences for MIN to the baseline are significantly affected.

For $\lambda = 0.5$, the relevance-only reference `Top` achieves SUM objective scores comparable to the baseline, and for MIN its score is even higher. As expected, for higher λ values the `Top` scores deteriorate soon. We also observe that our incremental algorithms are able to even diversify the top-k results for $\lambda < 1.0$ and always outperform the other variants (except for one case, MIN at $k = 50$ and $\lambda = 2.0$).

Performance Results The runtime of our experiments is shown in Figure 7.3. Note that all results are shown with log-log scales. In this representation, all polynomial complexity classes are displayed as straight lines, with varying slopes according to the degree of complexity, *i. e.*, $O(N^2)$ has a steeper slope than $O(N)$. Each individual run is represented by a corresponding dot. In addition, we show the trend line for these results. All experimental results correspond to the analytical predictions presented in Section 7.3.4. We see that the extracted queries yield result set sizes from 100 up to 10,000 documents. As expected, processing times of the non-incremental algorithms grow quadratic with this set size.

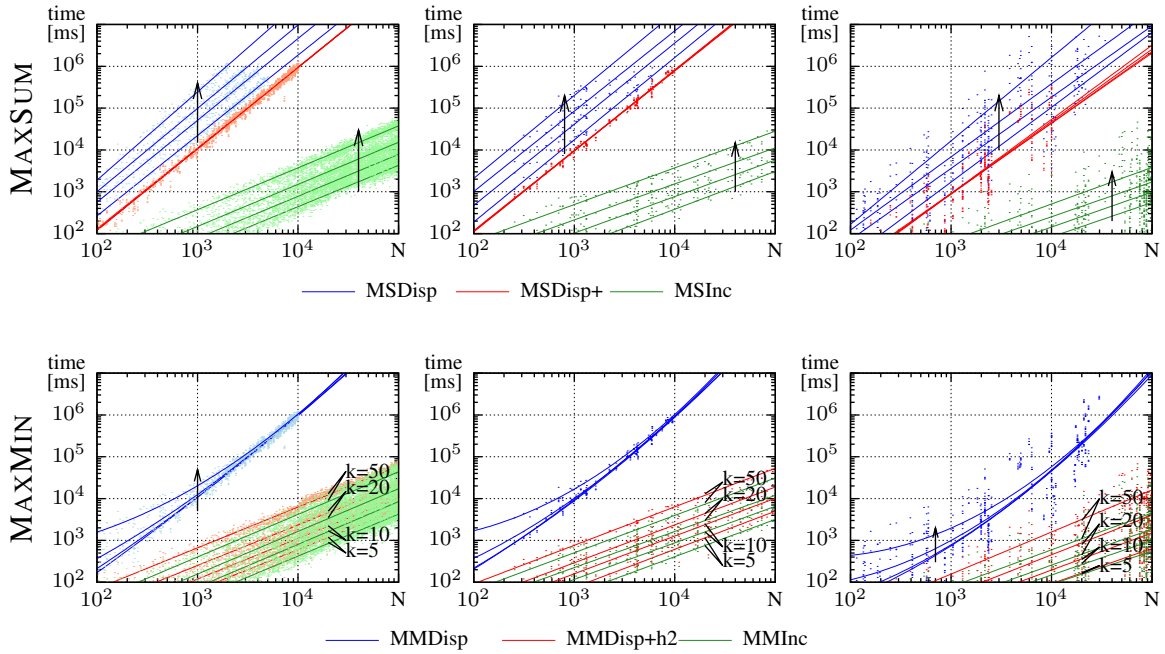


Figure 7.3: Runtime for MaxSum (top row) and MaxMin (bottom row) diversification algorithms on the three data sets NYTimes, Blogs08, and DBpedia (left to right) for $k = 5, 10, 20, 50$.

Computing the distance values for our document collections NYTimes and Blogs08 is equally fast for all documents. In contrast, we observe a high variance for the DBPedia collection (Figure 7.3, right column). This can be explained by characteristics of the distance function used for the structured data. The performance of this function depends on the number of attributes which varies by an order of magnitude in our query set. Regardless of this variance, the fitting curve matches the analytical predictions also for this data set.

Already for N = 1,000 and k = 10, `MSDisp` requires approximately one minute to compute the diversified subset. In contrast, our incremental variant needs only around 100 milliseconds for this set size. `MMInc` has slightly higher response times than `MSInc`, due to the additional k factor in objective computation. The figures also show the influence of k on the performance. Summarized, the incremental algorithms significantly outperform their baselines for set sizes larger than 1,000; usually by several orders of magnitude.

7.4.3 Stream Diversification

As a second experiment, we evaluate our algorithms against the *stream diversification* task. The experimental setting mimics a stream of results as retrieved from a news or blogs publish / subscribe-system, for example a RSS stream. In contrast to the set-based experiments, the items are ordered by time stamp instead of relevance score.

Algorithms With Equation 7.5 we define the task of stream diversification as finding at each position of the stream a diverse subset of all items that are observed until that position. As the baseline algorithm, we take all these observed results as a set and obtain the diverse set with the set-based baseline algorithms. This computation turn out to be very expensive. We are unable to process the baseline for streams larger than 10,000 while collecting a sufficient amount of measurement points. For that amount, the diversification requires nearly one hour, although we chose k = 10 to keep the computational cost small.

Additional to the baseline algorithm, we evaluate a window-variant following the work of Drosou and Pitoura [68] on stream diversification, denoted as `MSDispWin` and `MMDispWin`, respectively. We show the results for a jumping window of 100. Note that neither the algorithm proposed in [68] nor our window-based dispersion variants can be modified to diversify efficiently over the complete stream.

As before, we show two reference algorithms for comparison purposes. The `Last` reference algorithm simply proposes the k most recent items from a stream as the diverse set, whereas `TopRel` picks the k most relevant items from the stream, completely ignoring diversity.

Optimally, we would compute the diverse set after each new item. However, this is prohibitively expensive for the baseline. Therefore, we distribute ten measuring positions logarithmically over the stream, and compute the results for all algorithms at these positions.

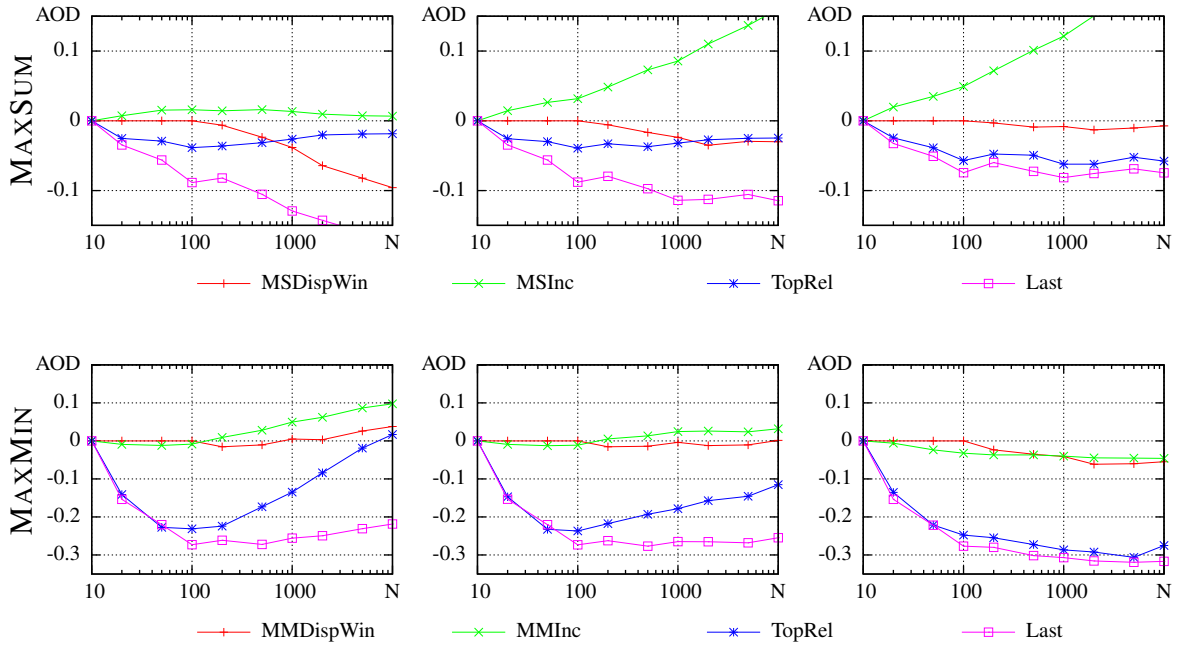


Figure 7.4: Absolute objective improvements on the Blogs08 data set of MAXSUM (top row) and MAXMIN (bottom row) with $\lambda = 1.0$ for streamline with no decay and decay over $100k$ and $1k$ (columns from left to right).

Results Figure 7.4 shows diversification quality for stream data, computed on Blogs08, our largest data set. To evaluate the influence of item relevance decay, the experiments are performed without decay (left column) and with a half-life over 1,000 (middle) and 10 (right) items.

As expected, `Last` yields very low objective scores, and `TopRel` is only slightly better on average. Both `MSInc` / `MMInc` and `MSDispWin` / `MMDispWin` achieve diversification scores comparable to the baseline, with a slight advantage for the incremental variants.

With respect to performance, the baseline algorithm is unusable in practice to perform stream diversification, as mentioned above. `MSDispWin` and `MMDispWin` require on average 250 ms and 80 ms per step. `MSInc` and `MMInc` update the diverse set for each incoming item in virtually no time (below time measure accuracy of 1 ms).

We can conclude that the windows-based as well as the incremental variants are efficient enough for stream diversification. `MSInc` and `MMInc` both have the advantage that the final outcome can immediately be updated for each new incoming item; this would be substantially more expensive for `MSDispWin` and `MMDispWin`.

7.5 Conclusion

Diversification is an important strategy to improve user satisfaction in the presence of ambiguous or broad queries. Due to their linear memory complexity, current approximation algorithms cannot be used to diversify large sets of items, or a large number of sets concurrently. Our new incremental approach is linear in computational complexity and constant in memory complexity *w. r. t.* the number of items, and therefore also very well suited for very large sets. As a framework for incremental diversification, our approach is applicable to a wide range of diversification objectives. A thorough evaluation based on three real-world data sets shows that such an incremental computation exhibits the same quality as the baseline algorithm. On the other hand, computation and thus query response time is sped up by several orders of magnitude for sets of a few thousand items and more. In addition, our streaming-based computation enables very efficient diversification of queries over continuous data sources as well, such as news streams or tweets, opening up further applications of diversification.

Chapter 8

Summary, Outlook, and Conclusion

This thesis addresses the question how users can be supported in accessing the ever growing amount of information stored on their computer Desktops. I primarily focus on aspects that regard the semantics of information, which are not supported by the classical Desktop metaphor as well as classical Desktop search engines. The semantification of the Desktop is the key to achieve better quality of both, the search process and the information being searched for. The main objectives throughout this thesis are to improve the efficiency of such systems and the convenience for users, with a focus on practical solutions and prototypical implementations.

To deliver a convenient and efficient Semantic Desktop search experience, improvements at each layer of such a system are investigated in this thesis — from data extraction and indexing to query articulation and evaluation. In particular, this thesis makes the following contributions:

1. I propose a Semantic Desktop search engine infrastructure that reuses major building blocks of a mature classical Desktop search engine, provides fundamental semantic services such as an RDF store, and allows for easy integration of new semantic services. Such services are metadata filters, metadata extraction modules, and semantic standalone applications that directly access the semantic store, *e. g.*, our RDF Visualizer.

2. An easy yet powerful structured query construction model and its graphical prototype tackles the drawbacks of search facilities of existing systems. It allows for hybrid query articulation but does not require knowledge of any query language or data ontology.

3. Our SPARQL-compliant RDF store LuceneSail provides an emancipated integration of semantic and full-text search that facilitates full potential of both worlds. In contrast, all major RDF store implementations only exhibit a shallow integration of full-text search. Additionally, I achieve performance improvements for the structured search part of hybrid queries regarding query evaluation planning and on-disk data structure optimization.

4. With the first benchmark focusing at full-text search capabilities and performance of RDF stores, I contribute a tool that allows semantic application and store developers to evaluate and compare feature richness and performance of semantic stores. I further provide a comparative study of the four major open-source RDF stores.

5. I propose a generic framework for incremental diversification of large sets and streams of search results and present incremental variants of two commonly used diversification objectives. These algorithms exhibit improved memory and runtime performance, while diversification quality is comparable to state-of-the-art approximation algorithms.

In the following, I highlight details of these contributions together with a respective outlook.

Semantic Desktop Search

With my contribution to the NEPOMUK Social Semantic Desktop project, in particular the central RDF repository and the semantic search module, we published a semantic infrastructure that allows for easy integration of semantic applications into the NEPOMUK Desktop. This infrastructure bases on Web standards such as RDF, RDFS, and SPARQL, is well documented and publicly available as open source software[1].

As a semantic application for the Semantic Desktop, we developed a Semantic Desktop Search engine that bases on an established classical Desktop search system and allows for modular extension with semantic services. With my development of a flexible architecture and my contributions to various semantic services as the metadata filters, metadata enrichment modules, semantic search capability, result visualization, and metadata browsing, I could facilitate a number of research activities that base on this system [47–49, 64, 177, 178]. We further show with a user study that semantification of Desktop search improves user satisfaction and search quality [122].

Finally, we gained significant influence on the open source Desktop Search community regarding introducing Semantic Web Technologies. There are three projects to name in particular:

1. The Gnome Beagle project — the one Beagle^{++} bases on — later on added a mapping from its structured data to RDF and a statement-based query interface.

2. The KDE Desktop is shipped with a semantic infrastructure and NEPOMUK ontologies.

3. The relatively new Tracker Desktop Search project[2] also builds on these ontologies and provides a central repository that can be used by applications to share their data which just opens their information silos.

When we look beyond the Semantic Desktop, we see that Desktop search must be extended to all areas where personal information is stored. In recent years, we witnessed the evolvement of the Social Web (Web 2.0), in which user-generated content and user-to-user interaction supersede the classical flow direction of information. All the information that is generated by a user and distributed over the Web constitutes an online entailment of the user's Desktop and is therefore personal information. This should also become available via Semantic Desktop Search and is therefore a relevant future research direction.

We further consider a *distributed* Semantic Desktop infrastructure an interesting open research question that fits a different setting (but nevertheless relevant) than the semantic infrastructure used in our work. In such a distributed system, each application may maintain its own semantic store and may provide a query interface to other applications. These stores require a distributed discovery and search facility as it is currently developed for the Semantic Web [148]. However, the particular setting of a user Desktop to inhabit a distributed Semantic Web poses challenges and opportunities to such a technology that constitutes a future direction.

[1] The NEPOMUK Social Semantic Desktop: `http://dev.nepomuk.semanticdesktop.org/wiki/EclipseDevelopment`.

[2] Gome Tracker: `http://projects.gnome.org/tracker/`.

Structured Query Articulation

With today's established keyword search interfaces, rich semantics as they are provided by a semantic infrastructure cannot fully or at least only implicitly be exploited by the user. Thus means to explicitly articulate rich structured queries needed to be studied.

Our structured query construction model enables users to construct arbitrarily complex semantic queries by using only two fundamental operations:

1. Typing a node or relation.

2. Adding a new relation to a node.

Our prototype implementation provides a graphical user interface that employs the ontology of existing data to provide the user with proper ontological terms. By this, the user cannot mistakenly violate the schema. This avoids empty result sets and user frustration. A recommendation architecture allows to employ different algorithms that aim at intelligently delivering proper terms to the user. Such semantic queries can then be translated into syntactically and ontologically valid queries using various structured query languages (such as SPARQL), which can directly be processed by any RDF store.

With this structured query builder, we close the gap between the semantic information extracted from user's Desktop data and stored in the RDF repository, and the semantic information need in user's mind. The user quickly learns the structure of the knowledge that exists on her Semantic Desktop and can easily express and refine a query. The query builder is publicly available as it is shipped with the NEPOMUK Semantic Desktop Rich Client[3].

With such a user interface, studies can now be conducted to learn how much structure users can express in a query given that the search engine understands the structure. An observation that users prefer simple types of queries such as keywords should be challenged by the hypothesis that people refrain from using complex queries because they do not expect search engines to properly understand structural information. In such a case, people would suspect the additional information to decrease the quality of search results.

But if people learn that structural information is correctly understood by the system, rich queries should be facilitated more frequently in cases where simple queries are insufficient. Additionally, one could also try to identify those user groups that benefit most from complex queries in order to have them more frequently employ structured query interfaces in their everyday life.

Evaluating Hybrid Queries

With such a search interface, complex hybrid queries that contain keywords referring to resources can be articulated. Such a class of structured queries pose a challenge to RDF stores. We identified and exploited performance improvement potential, which is necessary in order to provide users with enjoyable semantic search performance.

[3]NEPOMUK Eclipse: `http://nepomuk-eclipse.semanticdesktop.org/`.

The incorporation of full-text search into semantic search and a number of performance improvements provide a Semantic Web standard compliant hybrid query solution demanded by semantic application developers. The improvements target at different levels of structured query evaluation:

1. cardinality statistics and query evaluation planning
2. on-disk data structures
3. index selection strategy

These improvements allow for efficient evaluation of complex hybrid queries in multiple semantic applications [122, 131, 165] and to support research activities [46, 64, 168] as well as commercial products by Aduna Autofocus and ABC News[4]. The latter uses our system to base many features across their site, which as of August 2009 rose with over 200 million page views per month to the 5th rank in Nielsen rankings of online news sites[5].

We believe that hybrid query evaluation will benefit from a holistic theory that integrates structured search with full-text search, a theory that combines (Relational or Graph) Database systems with Information Retrieval systems.

Benchmarking Hybrid Queries

Today, semantic application developers can choose among various RDF store implementations to provide semantic full-text search; one of the most common search functionality of such applications. Unfortunately, the functionality and performance varies among such stores. In order to make a sophisticated decision, developers have to compare available RDF store implementations. However, no publicly available benchmark was available at that time that allowed for comparable and reproducible performance measurements.

With our work, we address this demand as we provide the first benchmark focusing at full-text search for semantic stores. With that, performance evaluations of hybrid queries become reproducible and comparable among stores for the first time. This benchmark contains a scalable synthetic data generator that is particularly suitable for structured full-text search. We further developed a set of 21 queries that stress different facets and features of hybrid query evaluation. Our benchmark enables semantic application developers to identify those stores that best fit their performance and feature requirements, as well as semantic store developers to identify particular bottlenecks and missing features.

In our analysis, we further identified the lack of today's semantic stores to support complex hybrid queries — hybrid queries that contain multiple full-text queries. Several of our research activities expose such a requirement and could there benefit from this performance analysis.

[4] ABC News: `http://abcnews.go.com/`.

[5] According to e-mail conversation with an ABC News software developer on Aug 17, 2009.

Diversifying Search Results

The last facet of semantic search that this work contributes to is the field of search result diversification. Here, a small set of relevant and diverse results are to be chosen from a large result list. This allows to easily provide the user with a representative overview of all available results, which improves user satisfaction and reduces the risk of user saturation.

In case of large result sets or streams of results — as they exist for publish/subscribe systems or continuous queries on news streams and tweets — no satisfying performance is provided by state-of-the-art diversification approaches. Those exhibit large memory requirements or super-linear runtime performance, which both are improved by our algorithms.

As a first contribution, we formalize the diversification task as a relaxation of the known set-diversification problem. We develop an incremental diversification framework and two incremental variants of the most commonly used diversification objectives: MAXSUM and MAXMIN. With these specific instances, linear runtime and constant memory complexity could be achieved. An extensive evaluation on three different real-world data sets shows that generated sets are similarly diverse compared to state-of-the-art diversification algorithms.

An aspect that our evaluation does not cover, as it is generally not yet considered in the literature, is the correlation between relevance and diversity of results. The questions such an analysis answers is whether queries can be categorized into different classes of correlation, and whether particular algorithms are better suited for one or the other class. For instance, a query with a strong correlation would not benefit from diversification at all, whereas for queries with no such correlation or even negative correlation, diversification should work very well. Such a classification of queries is not considered in literature where systems are evaluated over workloads of queries that potentially belong to different correlation classes.

In the light of stream diversification, such a relevance-diveristy correlation analysis has an additional dimension: time. For queries with diversity increasing over time while equally relevant results are seen, an algorithm that focuses on most recent results should perform much better than for queries where diverse results are seen early in the stream while the coverage of the topic degrades to a few aspects over time. In news and blog media, a query diversity classification could help to better understand how diversification algorithms perform.

Conclusion

We believe that semantic technologies will play a central role in managing information and knowledge on the Web in the near future. The adoption rate by developers and users rises continuously and semantic applications (Desktop and Web applications) soon will reach a critical mass to gain general public interest. Though a semantic killer application is not yet in sight, the technologies exist for the next step of evolution from the Web of user-generated content — Web 2.0 with killer applications like Facebook, Twitter, YouTube, and Wikis — towards the Web of user-generated knowledge — Web 3.0, the social Web of knowledge.

The Semantic Desktop will take a pivotal share in this evolution, with this thesis contributing to the development of high-performance and convenient-to-use Semantic Desktop Search.

Bibliography

[1] D. K. Agrafiotis and V. S. Lobanov. An efficient implementation of distance-based diversity measures based on k-d trees. *Journal of Chemical Information and Computer Sciences*, 39(1):51–58, 1999.

[2] R. Agrawal, S. Gollapudi, A. Halverson, and S. Ieong. Diversifying Search Results. In *Proceedings of the 2nd ACM International Conference on Web Search and Data Mining (WSDM'09)*, pages 5–14, New York, NY, USA, 2009. ACM.

[3] S. Agrawal, S. Chaudhuri, and G. Das. DBXplorer: a system for keyword-based search over relational databases. In *Proceedings of the 18th IEEE International Conference on Data Engineering (ICDE'02)*, San Jose, CA, USA, Feb. 26 – Mar. 1 2002. IEEE Computer Society.

[4] H. Alani. TGVizTab: An Ontology Visualisation Extension for Protégé. In *Workshop on Visualization Information in Knowledge Engineering (VIKE'03) at the 2nd International Conference on Knowledge Capture (K-Cap'03)*, Oct. 26 2003.

[5] S. Amer-Yahia, P. Case, T. Rölleke, J. Shanmugasundaram, and G. Weikum. Report on the DB/IR panel at the 2005 International ACM SIGMOD Conference on Management of Data (SIGMOD'05). *ACM SIGMOD Record*, 34(4):71–74, 2005.

[6] G. Antoniou and F. V. Harmelen. *A Semantic Web Primer*. MIT Press, Cambridge, MA, USA, 2nd edition edition, 2008.

[7] K. Anyanwu, A. Maduko, and A. P. Sheth. Semrank: Ranking complex relationship search results on the semantic web. In *Proceedings of the 14th International Conference on World Wide Web (WWW'05)*, pages 117–127, Chiba, Japan, May 10–14 2005. ACM.

[8] S. Auer and J. Lehmann. What have innsbruck and leipzig in common? extracting semantics from wiki content. In *Proceedings of the 4th European Semantic Web Conference (ESWC'07)*, pages 503–517. Springer, 2007.

[9] S. Babu and J. Widom. Continuous queries over data streams. *ACM SIGMOD Record*, 30(3):109–120, Sep. 2001.

[10] R. A. Baeza-Yates and M. P. Consens. The continued saga of db-ir integration. In *(e)Proceedings of the 30th International Conference on Very Large Data Bases (VLDB'04)*, page 1245, Toronto, Canada, Aug. 31 – Sep. 3 2004. Morgan Kaufmann.

[11] R. A. Baeza-Yates and G. H. Gonnet. Fast Text Searching for Regular Expressions or Automaton Searching on Tries. *Journal of the ACM*, 43(6):915–936, 1996.

[12] R. A. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. ACM Press / Addison-Wesley, 1999.

[13] A. Balmin, V. Hristidis, and Y. Papakonstantinou. Authority-Based Keyword Queries in Databases using ObjectRank. In *(e)Proceedings of the 30th International Conference on Very Large Data Bases (VLDB'04)*, 2004.

[14] D. F. Barbieri, D. Braga, S. Ceri, E. D. Valle, and M. Grossniklaus. C-sparql: Sparql for continuous querying. In *Proceedings of the 18th International Conference on World Wide Web*, pages 1061–1062, Madrid, Spain, April 20–24 2009.

[15] D. Barreau and B. A. Nardi. Finding and reminding: file organization from the desktop. *ACM SIGCHI Bulletin*, 27(3):39–43, 1995.

[16] H. Bast and I. Weber. The CompleteSearch engine: Interactive, efficient, and towards IR& DB integration. In *Proceedings of the 3rd Biennial Conference on Innovative Data Systems Research (CIDR'07)*, 2007.

[17] M. J. Bates. The design of browsing and berrypicking techniques for the online search interface. *Online Review*, 13(5):407–424, 1989.

[18] I. Becerra-Fernandez. Searching for experts on the web: A review of contemporary expertise locator systems. *ACM Transactions on Internet Technology*, 6(4):333–355, November 2006.

[19] D. Beckett and J. Grant. SWAD-Europe Deliverable 10.2: Mapping Semantic Web Data with RDBMSes. `http://www.w3.org/2001/sw/Europe/reports/scalable_rdbms_mapping_report/`, Jan. 2003.

[20] N. J. Belkin. *Interaction with texts: Information retrieval as information-seeking behavior*, volume 93, pages 55–66. Universitätsverlag Konstanz, 1993.

[21] J. V. D. Bercken. An evaluation of generic bulk loading techniques. In *(e)Proceedings of the 27th International Conference on Very Large Data Bases (VLDB'01)*, pages 461–470, 2001.

[22] M. K. Bergman. The deep web: Surfacing hidden value. *The Journal of Electronic Publishing*, 7(1), August 2001.

[23] T. Berners-Lee. *Weaving the Web: The Original Design and Ultimate Destiny of the World Wide Web*, volume first paperback edition. HarperCollins Publishers Inc., New York, NY, USA, 2000.

[24] T. Berners-Lee. Long live the web: A call for continued open standards and neutrality. *Scientific American*, Nov. 22 2010.

[25] T. Berners-Lee, R. Cailliau, and B. Pollermann. World wide web: The information universe. *Communications of the ACM*, 37:76–82, 1992.

[26] T. Berners-Lee, J. Hendler, and O. Lassila. The Semantic Web. *Scientific American*, 279(5), May 2001.

[27] A. Bernstein and E. Kaufmann. Gino - a guided input natural language ontology editor. In *Proceedings of the 5th International Semantic Web Conference (ISWC'06)*, pages 144–157, Athens, GA, USA, Nov. 5–9 2006. Springer.

[28] R. Bhagdev et al. Hybrid Search: Effectively Combining Keywords and Semantic Searches. In *Proceedings of the 5th European Semantic Web Conference (ESWC'08)*, pages 554–568, Tenerife, Canary Islands, Spain, June 1–5 2008.

[29] C. Bizer. The emerging web of linked data. *IEEE Intelligent Systems*, 24:87–92, 2009.

[30] C. Bizer and A. Schultz. Benchmarking the Performance of Storage Systems that expose SPARQL Endpoints. In *Proceedings of the 4th International Workshop on Scalable Semantic Web Knowledgebase Systems (SSWS'08)*, 2008.

[31] R. Boardman. *Improving Tool Support for Personal Information Management*. PhD thesis, Intelligent and Interactive Systems Group, Dept. of Electronic and Electrical Engineering, Imperial College London, UK, Sept. 2004.

[32] J. Borsje and H. Embregts. Graphical query composition and natural language processing in an rdf visualization interface. Bachelor thesis, Erasmus School of Economics and Business Economics, Rotterdam: Erasmus University, 2006.

[33] C. Boyle and K. Ratliff. A survey and classification of hypertext documentation systems. *IEEE Transactions on Professional Communication*, 35(2):98–111, June 1992.

[34] J. Broekstra, A. Kampman, and F. van Harmelen. Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema. In *Proceedings of the 2nd International Semantic Web Conference*, pages 54–68. Springer, 2002.

[35] V. Bush. As We May Think. *The Atlantic Monthly*, July 1945.

[36] J. G. Carbonell and J. Goldstein. The use of MMR, diversity-based reranking for reordering documents and producing summaries. In *Proceedings of the 20th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR'98)*, 1998.

[37] J. J. Carroll et al. Jena: Implementing the Semantic Web Recommendations. In *WWW Alternate track papers & posters*, pages 74–83, New York, NY, USA, 2004. ACM.

[38] B. Carterette. An analysis of NP-completeness in novelty and diversity ranking. In *ICTIR*, pages 200–211, 2009.

[39] M. Castells. *The rise of the network society*. Information age. Blackwell Publishers, 2000.

[40] T. Catarci, M. F. Costabile, S. Levialdi, and C. Batini. Visual query systems for databases: A survey. *Journal of Visual Languages and Computing*, 8(2):215–260, 1997.

[41] T. Catarci, P. Dongilli, T. D. Mascio, E. Franconi, G. Santucci, and S. Tessaris. An ontology based visual tool for query formulation support. In *Proceedings of the 16th European Conference on Artificial Intelligence*, 2004.

[42] F. Cesarini and G. Soda. An algorithm to construct a compact b-tree in case of ordered keys. *Inf. Process. Lett.*, 17(1):13–16, 1983.

[43] S. Chaudhuri et al. Integrating DB and IR Technologies: What is the Sound of One Hand Clapping? In *Proceedings of the 2nd Biennial Conference on Innovative Data Systems Research (CIDR'05)*, pages 1–12, Asilomar, CA, USA, January 4–7 2005.

[44] H. Chen and D. R. Karger. Less is more: probabilistic models for retrieving fewer relevant documents. In *Proceedings of the 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR'06)*, 2006.

[45] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. Niagaracq: a scalable continuous query system for internet databases. In *Proceedings of the 2000 International ACM SIGMOD Conference on Management of Data (SIGMOD'00)*, pages 379–390, Dallas, Texas, USA, May 16–18 2000. ACM. ISBN 1-58113-218-2.

[46] J. Chen, H. Guo, W. Wu, and W. Wang. imecho: an associative memory based desktop search system. In *Proceeding of the 18th ACM CIKM Conference on Information and Knowledge Management (CIKM'09)*, pages 731–740, New York, NY, USA, 2009. ACM.

[47] S. Chernov, G. Demartini, E. Herder, M. Kopycki, and W. Nejdl. Evaluating personal information management using an activity logs enriched desktop dataset. In *Proceedings of 3rd Personal Information Management Workshop (PIM'08)*, 2008.

[48] S. Chernov, E. Minack, and P. Serdyukov. Converting desktop into a personal activity dataset. In *Proceedings of 8th Nationational Russian Conference on Digital Libraries (RCDL'07)*, Pereslavl, Russia, Oct. 15–18 2007.

[49] S. Chernov, P. Serdyukov, P.-A. Chirita, G. Demartini, and W. Nejdl. Building a desktop search test-bed. In *Proceedings of the 29th European conference on IR research*, ECIR'07, pages 686–690, Berlin, Heidelberg, 2007. Springer-Verlag.

[50] P.-A. Chirita, S. Costache, W. Nejdl, and R. Paiu. Semantically Enhanced Searching and Ranking on the Desktop. In *Proceedings of the International Semantic Web Conference Workshop on The Semantic Desktop*, Galway, Ireland, November 2005.

[51] J. Cho. A fast regular expression indexing engine. In *Proceedings of the 18th IEEE International Conference on Data Engineering (ICDE'02)*, San Jose, CA, USA, Feb. 26 – Mar. 1 2002. IEEE Computer Society.

[52] K. W. Church and P. Hanks. Word Association Norms, Mutual Information, and Lexicography. *Conputational Linguistics*, 16(1):22–29, 1990.

[53] C. L. Clarke, M. Kolla, G. V. Cormack, O. Vechtomova, A. Ashkan, S. Büttcher, and I. MacKinnon. Novelty and diversity in information retrieval evaluation. In *Proceedings of the 31st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR'08)*, 2008.

[54] D. Comer. The ubiquitous b-tree. *ACM Computing Surveys*, 11(2):121–137, 1979.

[55] J. Conklin. A survey of hypertext. Technical Report STP-356-86, Microelectronics and Computer Technology Corporation (MCC), Austin, TX, USA, October 1986.

[56] D. Connolly, F. van Harmelen, I. Horrocks, D. L. McGuinness, P. F. Patel-Schneider, and L. A. Stein. DAML+OIL (March 2001) Reference Description, 18 December 2001. `http://www.w3.org/TR/daml+oil-reference`.

[57] M. P. Consens et al. XML Retrieval: DB/IR in Theory, Web in Practice. In *Proceedings of the 33th International Conference on Very Large Data Bases (VLDB'07)*, pages 1437–1438, University of Vienna, Austria, September 23–27 2007. ACM.

[58] C. A. Cutter. The buffalo public library in 1983. *Library Journal*, pages 211–217, 1883.

[59] M. C. Daconta, L. J. Obrst, and K. T. Smith. *The Semantiv Web: A Guide to the Future of XML, Web Services, and Knowledge Management*. Wiley Publishing, Inc., Indianapolis, IN, USA, 2003.

[60] A. Das Sarma, S. Gollapudi, and S. Ieong. Bypass rates: reducing query abandonment using negative inferences. In *Proceeding of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '08, pages 177–185, New York, NY, USA, 2008. ACM.

[61] S. F. C. de Araújo and D. Schwabe. Explorator: a tool for exploring rdf data through direct manipulation. In *Proceedings of the WWW'09 Workshop on Linked Data on the Web (LDOW'09)*, Madrid, Spain, 20–24th April 2009.

[62] S. Decker and M. Frank. The social semantic desktop. Technical Report 2004-05-02, DERI — Digital Enterprise Research Institute, May 2004.

[63] S. DeFazio et al. Integrating IR and RDBMS Using Cooperative Indexing. In *Proceedings of the 18th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '95)*, pages 84–92, Seattle, Washington, USA, July 9–13 1995. ACM Press.

[64] G. Demartini and C. Niederée. Finding Experts on the Semantic Desktop. In *Personal Identification and Collaborations: Knowledge Mediation and Extraction (PICKME 2008) Workshop at ISWC 2008*, Karlsruhe, Germany, October 2008.

[65] E. Demidova, P. Fankhauser, X. Zhou, and W. Nejdl. DivQ: diversification for keyword search over structured databases. In *Proceedings of the 33rd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR'10)*, 2010.

[66] L. Ding, T. W. Finin, A. Joshi, R. Pan, R. S. Cost, Y. Peng, P. Reddivari, V. Doshi, and J. Sachs. Swoogle: a search and metadata engine for the semantic web. In *Proceedings of the 13th ACM CIKM International Conference on Information and Knowledge Management (CIKM'04)*, pages 652–659, Washington, DC, USA, November 8–13 2004. ACM.

[67] M. Drosou and E. Pitoura. Comparing diversity heuristics. Technical Report TR-2009-05, Computer Science Department, University of Ioannina, Greece, 2009.

[68] M. Drosou and E. Pitoura. Diversity over continuous data. *IEEE Bulletin on Data Engineering*, 32(4):49–56, 2009.

[69] M. Drosou, K. Stefanidis, and E. Pitoura. Preference-aware publish/subscribe delivery with diversity. In *DEBS*, 2009.

[70] S. Dumais, E. Cutrell, J. Cadiz, G. Jancke, R. Sarin, and D. C. Robbins. Stuff I've Seen: A System for Personal Information Retrieval and Re-Use. In *Proceedings of the 26th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '03)*, 2003.

[71] A. Ebersbach, M. Glaser, and R. Heigl. *Social Web*. UVK Verlagsgesellschaft mbH, Konstanz, Germany, 2008.

[72] P. Eklund, N. Roberts, and S. Green. OntoRama: Browsing an RDF Ontology using a Hyperbolic-like Browser. In *Proceedings of the 1st International Symposium on Cyber Worlds (CW'02)*. IEEE Computer Society, 2002.

[73] V. Ercegovac et al. The TEXTURE Benchmark: Measuring Performance of Text Queries on a Relational DBMS. In *Proceedings of the 31th International Conference on Very Large Data Bases (VLDB'05)*, pages 313–324, Trondheim, Norway, 2005.

[74] B. Fallenstein and T. J. Lukka. Hyperstructure: Computers built around things that you care about. `http://fenfire.org/manuscripts/2004/hyperstructure/`, June 30 2004.

[75] H. Garcia-Molina, J. D. Ullman, and J. D. Widom. *Database Systems: The Complete Book*. Prentice Hall, 2002.

[76] J. Gaugaz, E. Ioannou, C. Kohlschütter, E. Minack, L. Sauermann, and M. Sogrin. Local Search and Context Services. NEPOMUK Deliverable D2.2, 20th December 2007.

[77] J. Gemmell, G. Bell, R. Lueder, S. Drucker, and C. Wong. Mylifebits: fulfilling the memex vision. In *Proceedings of the 10th ACM International Conference on Multimedia (MULTIMEDIA'02)*, pages 235–238, New York, NY, USA, 2002. ACM.

[78] F. Giunchiglia. Managing Diversity in Knowledge. In M. Ali and R. Dapoigny, editors, *IEA/AIE 2006, LNAI 4031*, page 1. Springer-Verlag Berlin Heidelberg, 2006.

[79] S. Gollapudi and A. Sharma. An Axiomatic Approach for Result Diversification. In *Proceedings of the 18th International Conference on World Wide Web (WWW'09)*, pages 381–390, Madrid, Spain, 2009. ACM.

[80] J. Gray, editor. *The Benchmark Handbook For Database and Transaction Processing Systems*. Morgan Kaufmann, 1993.

[81] W. O. W. Group. OWL Web Ontology Language Document Overview, 2009. `http://www.w3.org/TR/owl-overview/`.

[82] W. R. C. W. Group. RDF Vocabulary Description Language 1.0: RDF Schema, Feb 2004. `http://www.w3.org/TR/rdf-schema/`.

[83] W. R. C. W. Group. Resource Description Framework (RDF), Feb 2004. `http://www.w3.org/RDF/`.

[84] R. V. Guha and R. McCool. Tap: a semantic web platform. *Computer Networks*, 42(5):557–577, 2003.

[85] A. Gulli and A. Signorini. The indexable web is more than 11.5 billion pages. In *Proceedings of the 14th International Conference on World Wide Web, Special Interest Tracks and Posters*, pages 902–903, Chiba, Japan, May 10–14 2005.

[86] Y. Guo, Z. Pan, and J. Heflin. LUBM: A Benchmark for OWL Knowledge Base Systems. *Journal of Web Semantics*, 3(2):158–182, 2005.

[87] J. R. Haritsa. The KNDN Problem: A Quest for Unity in Diversity. *IEEE Bulletin on Data Engineering*, 32(4):15–22, 2009.

[88] S. Harris and N. Gibbins. 3store: Efficient Bulk RDF Storage. In *Proceedings of the First International Workshop on Practical and Scalable Semantic Systems*, Sanibel Island, Florida, USA, 2003.

[89] A. Harth and S. Decker. Optimized Index Structures for Querying RDF from the Web. In *Proceedings of the 3rd Latin American Web Congress*, pages 71–80. IEEE Press, 2005.

[90] A. Harth, S. R. Kruk, and S. Decker. Graphical representation of rdf queries. In *Proceedings of the 15th International Conference on World Wide Web (WWW'06)*, pages 859–860, Edinburgh, Scotland, UK, May 23–26 2006. ACM.

[91] A. Harth, J. Umbrich, A. Hogan, and S. Decker. YARS2: A Federated Repository for Querying Graph Structured Datafrom the Web. In *Proceedings of the 6th International Semantic Web Conference (ISWC'07) and the 2nd Asian Semantic Web Conference (ASWC'07)*, pages 211–224, Busan, Korea, November 11–15 2007. Springer.

[92] T. H. Haveliwala. Topic-sensitive pagerank: A context-sensitive ranking algorithm for web search. *IEEE Transactions on Knowledge and Data Engineering*, 15(4):784–796, 2003.

[93] J. Hendler. Agents and the semantic web. *Intelligent Systems, IEEE*, 16(2):30–37, Mar–Apr 2001.

[94] M. Hildebrand et al. An analysis of search-based user interaction on the Semantic Web. Report, Centrum voor Wiskunde en Informatica (CWI), 2007. INS-E0706, ISSN 1386-3681.

[95] A. Hogan, A. Harth, J. Umbrich, S. Kinsella, A. Polleres, and S. Decker. Searching and Browsing Linked Data with SWSE: The Semantic Web Search Engine. Technical Report 2010-07-23, DERI, Jul. 2010.

[96] P. Howard and S. Jones. *Society online: the Internet in context*. Sage, 2004.

[97] V. Hristidis et al. Efficient IR-Style Keyword Search over Relational Databases. In *Proceedings of the 29th International Conference on Very Large Data Bases (VLDB'03)*, pages 850–861, 2003.

[98] V. Hristidis, O. Valdivia, M. Vlachos, and P. S. Yu. A system for keyword search on textual streams. In *Proceedings of the 7th SIAM International Conference on Data Mining (SDM'07)*, Minneapolis, MN, USA, Apr. 26–28 2007.

[99] C. Hölscher and G. Strube. Web search behavior of internet experts and newbies. *Computer Networks*, 33(1-6):337–346, 2000.

[100] Y. E. Ioannidis. Query Optimization. *ACM Comput. Surv.*, 28(1):121–123, 1996.

[101] E. Ioannou, C. Niederée, and W. Nejdl. Probabilistic Entity Linkage for Heterogeneous Information Spaces. In *CAiSE*, 2008.

[102] U. Irmak, S. Mihaylov, T. Suel, S. Ganguly, and R. Izmailov. Efficient query subscription processing for prospective search engines. In *USENIX Annual Technical Conference*, 2006.

[103] R. Jain. *The Art of Computer System Performance Analysis*. Wiley Professional Computing, Littleton, MA, USA, 1991.

[104] K. Jordan, J. Hauser, and S. Foster. The augmented social network: Building identity and trust into the next-generation internet. *First Monday*, 8(8), Aug. 4 2003.

[105] V. Kaptelinin and M. Czerwinski, editors. *Beyond the Desktop Metaphor*. MIT Press, Cambridge, MA, USA, 2007.

[106] E. Kaufmann and A. Bernstein. How useful are natural language interfaces to the semantic web for casual end-users. In *Proceedings of the 6th International Semantic Web Conference (ISWC'07) and the 2nd Asian Semantic Web Conference (ASWC'07)*, pages 281–294, Busan, Korea, November 11–15 2007. Springer.

[107] R. Kawase, E. Minack, W. Nejdl, S. Araújo, and D. Schwabe. Incremental End-user Query Construction for the Semantic Desktop. In *Proceedings of the 5th International Conference on Web Information Systems and Technologies (WEBIST'09)*, pages 270–275, Lisbon, Portugal, March 23–26 2009. INSTICC Press.

[108] J. M. Kleinberg. Authoritative Sources in a Hyperlinked Environment. *Journal of the ACM*, 46(5):604–632, 1999.

[109] J. Klonk. Comments on optimality of b-trees. *ACM SIGMOD Record*, 13(2):36–38, Jan. 1983.

[110] D. E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison Wesley, 1973.

[111] A. N. Langville and C. D. Meyer. Deeper inside pagerank. *Internet Mathematics*, 1(3):335–380, January 2004.

[112] R. Lee. Scalability Report on Triple Store Applications. `http://simile.mit.edu/reports/stores/`, July 2004.

[113] Y. Lei, V. S. Uren, and E. Motta. SemSearch: A Search Engine for the Semantic Web. In *15th International Conference on Knowledge Engineering and Knowledge Management Managing Knowledge in a World of Networks (EKAW 2006)*, 2006.

[114] F. Lelli, L. Sauermann, G. Å. Grimnes, M. Klinkigt, R. Stecher, R. Jäschke, K. Hajy, M. Kiesel, M. Loerscher, G. Reif, C. Bogdan, and E. Minack. NEPOMUK user guide. NEPOMUK Deliverable D6.7.A, 28th October 2008. `http://nepomuk.semanticdesktop.org/xwiki/bin/view/Main1/D6-7-A`.

[115] V. Lopez, V. Uren, E. Motta, and M. Pasin. Aqualog: An ontology-driven question answering system for organizational semantic intranets. *Journal of Web Semantics*, 5(2):72–105, 2007.

[116] S. Madden and M. J. Franklin. Fjording the stream: An architecture for queries over streaming sensor data. In *Proceedings of the 18th IEEE International Conference on Data Engineering (ICDE'02)*, pages 555–566, San Jose, CA, USA, Feb. 26 – Mar. 1 2002. IEEE Computer Society. ISBN 0-7695-1531-2.

[117] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to Information Retrieval*. Cambridge Univ Press, 2008.

[118] D. G. McDonald and J. Dimmick. The Conceptualization and Measurement of Diversity. *Communication Research*, 30(1):60–79, 2003.

[119] G. A. Miller. Wordnet: A lexical database for english. *Communications of the ACM*, 38(11):39–41, 1995.

[120] R. E. Miller, N. Pippenger, A. L. Rosenberg, and L. Snyder. Optimal 2,3-trees. *SIAM Journal of Computing*, 8(1):42–59, February 1979.

[121] E. Minack, G. Demartini, and W. Nejdl. Current Approaches to Search Result Diversification. In *Proceedings of The First International Workshop on Living Web at the 8th International Semantic Web Conference (ISWC'09)*, Chantilly, VA, USA, Oct. 25–29 2009.

[122] E. Minack, R. Paiu, S. Costache, G. Demartini, J. Gaugaz, E. Ioannou, P.-A. Chirita, and W. Nejdl. Leveraging Personal Metadata for Desktop Search: The Beagle^{++} System. *Journal of Web Semantics*, 8(1):37–54, March 2010.

[123] E. Minack and L. Sauermann. Adapters, Extractors, and Knowledge Structure Services. NEPOMUK Deliverable D2.1, 28th December 2006.

[124] E. Minack, L. Sauermann, G. Grimnes, C. Fluit, and J. Broekstra. The Sesame LuceneSail: RDF Queries with Full-text Search. Technical Report 2008-1, NEPOMUK, Feb 2008.

[125] E. Minack, W. Siberski, and W. Nejdl. Benchmarking Fulltext Search Performance of RDF Stores. In *Proceedings of the 6th European Semantic Web Conference (ESWC 2009)*, pages 81–95, Heraklion, Greece, 31th May – 4th June 2009.

[126] E. Minack, W. Siberski, and W. Nejdl. Incremental Diversification for Very Large Sets: a Streaming-based Approach. In *Proceedings of the 34th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR'11)*, Beijing, China, 24–28th July 2011.

[127] E. Minack, W. Siberski, G. Zenz, and X. Zhou. SUITS4RDF: Incremental Query Construction for the Semantic Web. In *Proceedings of the Poster & Demo Session at the 7th International Semantic Web Conference (ISWC2008)*, Karlsruhe, Germany, October 2008.

[128] G. Mishne and M. de Rijke. A study of blog search. In *ECIR*, 2006.

[129] T. M. Mitchell. Learning, information extraction and the web. In *Proceedings of the 11th European Conference on Principles and Practice of Knowledge Discovery in Databases (PKDD'07)*, page 1, Warsaw, Poland, Sep. 17–21 2007.

[130] T. M. Mitchell, J. Betteridge, A. Carlson, E. R. H. Jr., and R. C. Wang. Populating the semantic web by macro-reading internet text. In *Proceedings of the 8th International Semantic Web Conference (ISWC'09)*, pages 998–1002, Chantilly, VA, USA, Oct. 2009.

[131] K. Möller, S. Handschuh, T. Groza, G. Å. Grimnes, L. Sauermann, M. Jazayeri, E. Minack, C. Mesnage, G. Reif, and R. Guðjónsdóttir. The NEPOMUK Project — On the way to the Social Semantic Desktop. In *Proceedings of International Conferences on new Media technology (I-MEDIA-2007) and Semntic Systems (I-SEMANTICS-07)*, pages 201–210, Graz, Austria, September 2007.

[132] B. Motik, B. C. Grau, I. Horrocks, Z. Wu, A. Fokoue, and C. Lutz. OWL2 Web Ontology Language: Profiles. `http://www.w3.org/TR/owl2-profiles/`, 2009.

[133] B. Motik, P. F. Patel-Schneider, and B. C. Grau. OWL2 Web Ontology Language: Direct Semantics. `http://www.w3.org/TR/owl2-direct-semantics/`, 2009.

[134] W. Nejdl and R. Paiu. Desktop search - how contextual information influences search results & rankings. In *Proceedings of the ACM SIGIR 2005 Workshop on Information Retrieval in Context (IRiX)*, Salvador, Brazil, 2005.

[135] W. Nejdl and R. Paiu. I know i stored it somewhere - contextual information and ranking on our desktop. In *8th International Workshop of the EU DELOS Network of Excellence on Future Digital Library Management Systems*, March 2005.

[136] W. Nejdl, B. Wolf, C. Qu, S. Decker, M. Sintek, A. Naeve, M. Nilsson, M. Palmér, and T. Risch. Edutella: a p2p networking infrastructure based on rdf. In *Proceedings of the 11th international conference on World Wide Web*, WWW '02, pages 604–615, New York, NY, USA, 2002. ACM.

[137] T. H. Nelson. Complex information processing: a file structure for the complex, the changing and the indeterminate. In *Proceedings of the 20th National Conference*, ACM '65, pages 84–100, New York, NY, USA, 1965. ACM.

[138] T. H. Nelson. *Computer Lib: You can and must understand computers now / Dream Machines: New freedoms through computer screens — a minority report*. Microsoft Press, Redmond, WA, USA, rev. edition, 1987. ISBN 0-914845-49-7.

[139] M. E. J. Newman. Power laws, Pareto distributions and Zipf's law. *Contemporary Physics*, 46:323–351, Sep 2005.

[140] K. Ono and G. M. Lohman. Measuring the complexity of join enumeration in query optimization. In *Proceedings of the 16th International Conference on Very Large Databases (VLDB'90)*, pages 314–325, San Francisco, CA, USA, 1990. Morgan Kaufmann Publishers Inc.

[141] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford Digital Library Technologies Project, 1998.

[142] D. S. Parker, R. R. Muntz, and H. L. Chau. The tangram stream query processing system. In *Proceedings of the 5th IEEE International Conference on Data Engineering (ICDE'89)*,

pages 556–563, Los Angeles, CA, USA, February 6–10 1989. IEEE Computer Society. ISBN 0-8186-1915-5.

[143] J. Pérez, M. Arenas, and C. Gutierrez. nSPARQL: A Navigational Language for RDF. In *Proceedings of the 7th International Semantic Web Conference (ISWC 2008)*, pages 66–81, October 26–30 2008.

[144] E. Pietriga. IsaViz: A Visual Environment for Browsing and Authoring RDF Models. In *Proceedings of the 11th International Conference on World Wide Web (WWW'02)*, 2002.

[145] A.-M. Popescu. *Information Extraction from Unstructured Web Text*. PhD thesis, University of Washington Ph.D. Dissertation, 2007.

[146] E. Prud'hommeaux and A. Seaborne. SPARQL Query Language for RDF, Jan 2008. `http://www.w3.org/TR/rdf-sparql-query/`.

[147] D. Quan, D. Huynh, and D. R. Karger. Haystack: A platform for authoring end user semantic web applications. In *Proceeding of the Second International Semantic Web Conference*, pages 738–753, Sanibel Island, FL, USA, 203.

[148] B. Quilitz and U. Leser. Querying distributed rdf data sources with sparql. In *Proceedings of the 5th European Semantic Web Conference (ESWC'08)*, pages 524–538, Tenerife, Canary Islands, Spain, June 1–5 2008.

[149] A. Reggiori, D.-W. van Gulik, and Z. Bjelogrlic. Indexing and retrieving Semantic Web resources: the RDFStore model. SWAD-Europe Workshop. `http://www.w3.org/2001/sw/Europe/events/20031113-storage/positions/asemantics.html`, October 2003.

[150] G. Reif, T. Groza, S. Scerri, and S. Handschuh. Final NEPOMUK Architecture. NEPOMUK Deliverable D6.2.B, 4th December 2008. `http://nepomuk.semanticdesktop.org/xwiki/bin/view/Main1/D6%2D2%2DB`.

[151] C. Rocha, D. Schwabe, and M. P. de Aragao. A Hybrid Approach for Searching in the Semantic Web. In *Proceedings of the 13th International Conference on World Wide Web (WWW'04)*, 2004.

[152] A. L. Rosenberg and L. Snyder. Minimal comparison 2,3-trees. *SIAM Journal of Computing*, 7(4):465–480, November 1978.

[153] A. L. Rosenberg and L. Snyder. Compact b-trees. In *Proceedings of the 1979 International ACM SIGMOD Conference on Management of Data SIGMOD'79*, pages 43–51, New York, NY, USA, 1979. ACM.

[154] A. L. Rosenberg and L. Snyder. Time- and space-optimality in b-trees. *ACM Transactions on Database Systems (TODS)*, 6(1):174–193, March 1981.

[155] A. Russell and P. R. Smart. NITELIGHT: A graphical editor for SPARQL queries. In *Proceedings of the 7th International Semantic Web Conference — Posters & Demos (ISWC'08)*, Karlsruhe, Germany, Oct. 26–30 2008.

[156] A. Russell, P. R. Smart, D. Braines, and N. R. Shadbolt. Nitelight: A graphical tool for semantic query construction. In *Semantic Web User Interaction Workshop (SWUI 2008)*, Florence, Italy, 5th April 2008.

[157] T.-W. Ryu, , T. wan Ryu, and C. F. Eick. A unified similarity measure for attributes with set or bag of values. In *RSDMGrC*, 1998.

[158] E. Sandhaus. The New York Times Annotated Corpus. Linguistic Data Consortium (LDC), Philadelphia, 2008. `http://www.ldc.upenn.edu/Catalog/CatalogEntry.jsp?catalogId=LDC2008T19`.

[159] R. L. T. Santos, C. Macdonald, and I. Ounis. Exploiting query reformulations for web search result diversification. In *Proceedings of the 19th International Conference on World Wide Web (WWW'10)*, 2010.

[160] R. L. T. Santos, C. Macdonald, and I. Ounis. Exploiting query reformulations for web search result diversification. In *Proceedings of the 19th International Conference on World Wide Web*, pages 881–890, Raleigh, NC, USA, April 26–30 2010. ISBN 978-1-60558-799-8.

[161] L. Sauermann. The Gnowsis — Using Semantic Web Technologies to build a Semantic Desktop. Diploma thesis, Technische Universität Wien, 2003.

[162] L. Sauermann. *The Gnowsis Semantic Desktop approach to Personal Information Management*. PhD thesis, Fachbereich Informatik der Universität Kaiserslautern, June 2009.

[163] L. Sauermann, A. Bernardi, and A. Dengel. Overview and Outlook on the Semantic Desktop. In *Proceedings of the 4th International Semantic Web Conference (ISWC'05)*, Galway, Ireland, November 6–10 2005. Springer.

[164] L. Sauermann, G. Å. Grimnes, M. Kiesel, C. Fluit, H. Maus, D. Heim, D. Nadeem, B. Horak, and A. Dengel. Semantic desktop 2.0: The gnowsis experience. In *Proceedings of the 5th International Semantic Web Conference (ISWC'06)*, pages 887–900, Athens, GA, USA, Nov. 5–9 2006. Springer.

[165] L. Sauermann, G. Å. Grimnes, M. Kiesel, H. Maus, D. Heim, D. Nadeem, B. Horak, and A. Dengel. Semantic Desktop 2.0: The Gnowsis Experience. In *Proc. of the ISWC Conference*, pages 887–900, Nov 2006.

[166] L. Sauermann, E. Minack, M. Sogrin, C. Kohlschütter, and R. Kawase. Using the Personal Semantic Desktop. NEPOMUK Deliverable D2.3, 20th October 2008. `http://nepomuk.semanticdesktop.org/xwiki/bin/view/Main1/D2%2D3`.

[167] C. Sayers. Node-centric RDF Graph Visualization. Technical Report HPL-2004-60, HP Labs, 2004.

[168] S. Schenk, C. Saathoff, S. Staab, and A. Scherp. Semaplorer - interactive semantic exploration of data and media based on a federated cloud infrastructure. *Journal of Web Semantics*, 2009. accepted for publication.

[169] M. Schmidt et al. An Experimental Comparison of RDF Data Management Approaches in a SPARQL Benchmark Scenario. In *Proceedings of the 7th International Semantic Web Conference (ISWC'08)*, Karlsruhe, Germany, Oct. 26–30 2008.

[170] A. Seaborne. Rdql - a query language for rdf, Jan 2004. `http://www.w3.org/Submission/RDQL/`.

[171] T. Segaran, C. Evans, and J. Taylor. *Programming the Semantic Web*. O'Reilly Media, Inc., Sebastopol, CA, USA, 1 edition, 21 July 2009.

[172] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access Path Selection in a Relational Database Management System. In *Proceedings of the 1979 International Conference on Management of Data (SIGMOD'79)*, pages 23–34, 1979.

[173] M. Sintek, L. van Elst, S. Scerri, and S. Handschuh. Distributed Knowledge Representation on the Social Semantic Desktop: Named Graphs, Views and Roles in NRL. In *Proceedings of the 4th European Semantic Web Conference (ESWC 2007)*, pages 594–608, 2007.

[174] D. Skoutas, E. Minack, and W. Nejdl. Dealing with Diversity in Web Search Results. In *Proceedings of the 2nd ACM International Conference on Web Science (WebSci'10)*, Raleigh, NC, USA, April 26–27 2010.

[175] A. Slivkins, F. Radlinski, and S. Gollapudi. Learning optimally diverse rankings over large document collections. In *ICML*, 2010.

[176] W. Spillman and E. Lang. *The Law of Diminishing Returns*. World Book Company, Yonkers-on-Hudson, NY, USA, 1924.

[177] R. Stecher, C. Niederée, and W. Nejdl. Query rewriting for lightweight information integration. In *Proceedings of the Third International Conference on Digital Information Management (ICDIM'08)*, pages 375–380, Nov. 2008.

[178] R. Stecher, C. Niederée, and W. Nejdl. Wildcards for lightweight information integration in virtual desktops. In *Proceedings of the 17th ACM CIKM International Conference on Information and Knowledge Management (CIKM'08)*, pages 797–806, New York, NY, USA, 2008. ACM.

[179] M. Steyvers and T. Griffiths. *Latent Semantic Analysis: A Road to Meaning*, chapter Probabilistic Topic Models. Laurence Erlbaum, 2006.

[180] S. Tata and G. M. Lohman. SQAK: doing more with keywords. In *Proceedings of the 2008 International ACM SIGMOD Conference on Management of Data (SIGMOD'08)*. ACM, 2008.

[181] J. Teevan, C. Alvarado, M. Ackerman, and D. Karger. The Perfect Search Engine Is Not Enough: A Study of Orienteering Behavior in Directed Search. In *Proc. of CHI*, 2004.

[182] D. Terry, D. Goldberg, D. Nichols, and B. Oki. Continuous queries over append-only databases. In *Proceedings of the 1992 International ACM SIGMOD Conference on Management of Data (SIGMID'92)*, pages 321–330. ACM, 1992.

[183] E. Vee, U. Srivastava, J. Shanmugasundaram, P. Bhat, and S. A. Yahia. Efficient Computation of Diverse Query Results. In *Proceedings of the 24th IEEE International Conference on Data Engineering (ICDE'08)*, pages 228–236, Washington, DC, USA, 2008. IEEE Computer Society.

[184] W3C. Survey of RDF/Triple Data Stores. `http://www.w3.org/2001/05/rdf-ds/DataStore`, April 2003.

[185] H. Wang, K. Zhang, Q. Liu, T. Tran, and Y. Yu. Q2semantic: A lightweight keyword interface to semantic search. In *Proceedings of the 5th European Semantic Web Conference (ESWC 2008)*, pages 584–598, 2008.

[186] J. Wang and J. Zhu. Portfolio theory of information retrieval. In *Proceedings of the 32nd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR'09)*, 2009.

[187] H. C. Warren. *A History Of The Association Psychology*. Charles Scribner's Sons, 1921.

[188] D. Wood, P. Gearon, and T. Adams. Kowari: A Platform for Semantic Web Storage and Analysis. In *Conference Proceedings XTech 2005*, Amsterdam, Netherlands, 25–27 May 2005.

[189] A. Y. H. Xin Dong. A platform for personal information management and integration. In *Proceedings of the 2nd Biennial Conference on Innovative Data Systems Research (CIDR'05)*, pages 119–130, Asilomar, CA, USA, January 4–7 2005.

[190] R. M. Young. Association of Ideas. In P. P. Wiener, editor, *Dictionary of the History of Ideas*, volume 1, pages 111–118. Charles Scribner's Sons, New York, 1968.

[191] C. Yu, L. Lakshmanan, and S. Amer-Yahia. It takes variety to make a world: diversification in recommender systems. In *EDBT*, 2009.

[192] G. Zenz, X. Zhou, E. Minack, W. Siberski, and W. Nejdl. From Keywords to Semantic Queries—Incremental Query Construction on the Semantic Web. *Web Semantics: Science, Services and Agents on the World Wide Web*, 7(3):166–176, September 2009.

[193] C. Zhai, W. W. Cohen, and J. D. Lafferty. Beyond independent relevance: methods and evaluation metrics for subtopic retrieval. In *Proceedings of the 25th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR'03)*, New York, NY, USA, 2003. ACM.

[194] C. Zhai and J. D. Lafferty. A risk minimization framework for information retrieval. *Inf. Process. Manage.*, 42(1):31–55, 2006.

[195] L. Zhang, Q. Liu, J. Zhang, H. Wang, Y. Pan, and Y. Yu. Semplore: An IR Approach to Scalable Hybrid Query of Semantic Web Data. In *Proceedings of the 6th International Semantic Web Conference (ISWC'07) and the 2nd Asian Semantic Web Conference (ASWC'07)*, Busan, Korea, Nov. 11–15 2007. Springer.

[196] Q. Zhou, C. Wang, M. Xiong, H. Wang, and Y. Yu. SPARK: Adapting keyword query to semantic search. In *Proceedings of the 6th International Semantic Web Conference (ISWC'07)*, 2007.

[197] C.-N. Ziegler, S. M. McNee, J. A. Konstan, and G. Lausen. Improving Recommendation Lists through Topic Diversification. In *Proceedings of the 14th International Conference on World Wide Web (WWW'05)*, pages 22–32, New York, NY, USA, 2005. ACM.

[198] M. M. Zloof. Query-by-example: a database language. *IBM System Journal*, 16:324–343, 1977.

Wissenschaftlicher Lebenslauf

Name	Enrico Minack
Geboren am	15. Mai 1980
Geboren in	Eisenhüttenstadt

SCHULABSCHLUSS

9/1996 – 6/1999 **Allgemeine Hochschulreife**
Oberstufenzentrum "Gottfried Wilhelm Leibniz"
Eisenhüttenstadt

STUDIUM

10/2000 – 01/2006 **Angewandte Informatik**
Technische Universität Chemnitz
Informations- und Kommunikationssysteme

5/2005 – 11/2005 **Diplomarbeit**
Volkswagen AG Konzernforschung, Wolfsburg
"Evaluation of the influence of channel conditions on Car2X Communication"

PROMOTION

02/2006 – 12/2010 **Wissenschaftlicher Mitarbeiter und Doktorand**
Leibniz Universität Hannover
Forschungszentrum L3S